CSE 373: Data Structures & Algorithms

Balancing BSTs; Lazy Deletion; Amortized Analysis

Riley Porter

Winter 2017

Course Logistics

- HW1 due last night, HW2 (Asymptotic Runtime Analysis (Big-O) and Implementing Heaps) released tomorrow and due a week from Friday.
- Canvas does weird name things with resubmission, don't worry about it. We'll make future HW submissions a zip to avoid it.
- If you have weird technical issues with submitting HW, you can email your TA an attachment of your files. This shouldn't be the norm, but we can accept an email with that timestamp as a submission.
- Course message board will be better monitored, we have a schedule now.
- We'll be posting weekly summaries / self checks. Expect the first two week's posted tomorrow. They're just extra material for you to gauge how you're doing, feel free to ask questions about them in office hours or on the discussion board. This is hopefully to supplement not recording lectures in case you are nervous about what you need to have learned each week.
- Section materials will be posted online, the day before section, but solutions only available in section. If you have to miss a day, talk to your TA.

Topics from Last Lecture

- Floyd's algorithm for building a heap
  - proof of O(N) for buildHeap, which is faster than N **`inserts`** which would be O(NlogN)

- Review from 143 Dictionaries/Maps/Sets: understand how to be a client of them and the ADT, think about tradeoffs for implementations.
  - implementations all assuming non-hash structure

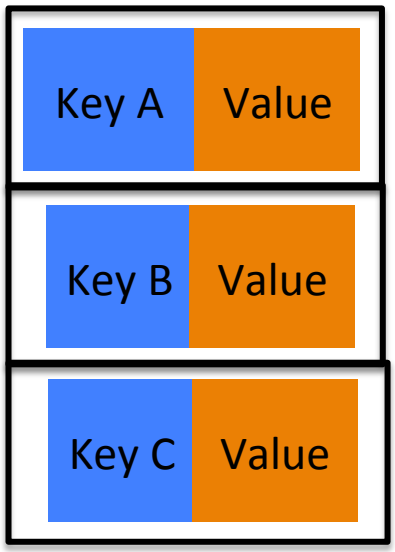
- Review from 143 Binary Search Trees: structure, insert, evaluate the runtime of operations. New thing: deleting from a BST. (findMin and findMax for node replacement)

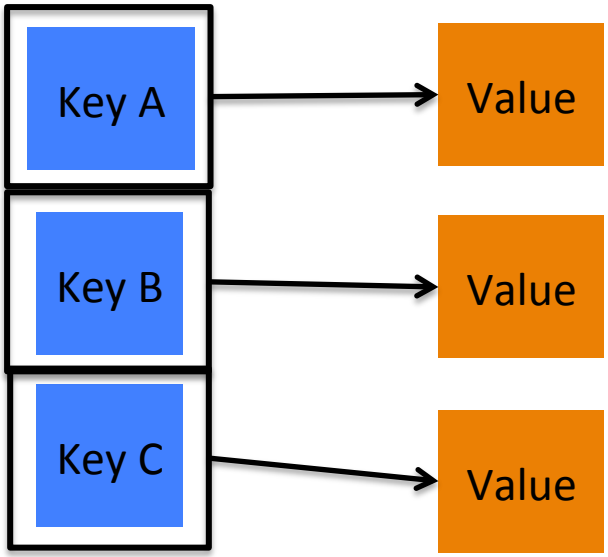
Review: Dictionary Implementation

We store the keys with their values so all we really care about is how the keys are stored.

– want fast operations for iterating over the keys

You could think about this in a couple ways:





Review: Simple Dictionary Implementations; Operation Analysis

For dictionary with *n* key/value pairs

	`insert`	`find`	`delete`
Unsorted linked-list	*O(1)**	*O(N)*	*O(N)*
Unsorted array	*O(1)**	*O(N)*	*O(N)*
Sorted linked list	*O(N)*	*O(N)*	*O(N)*
Sorted array	*O(N)*	*O(logN)*	*O(N)*

* Unless we need to check for duplicates

We'll see a Binary Search Tree (BST) probably does better, but not in the worst case unless we keep it balanced

Sorted Array: Lazy Deletion

10	12	24	30	41	42	44	45	50
✓	✗	✓	✓	✓	✓	✗	✓	✓

A *general technique* for making **`delete`** as fast as **`find`**:

- Instead of actually removing the item just mark it deleted

Plusses:

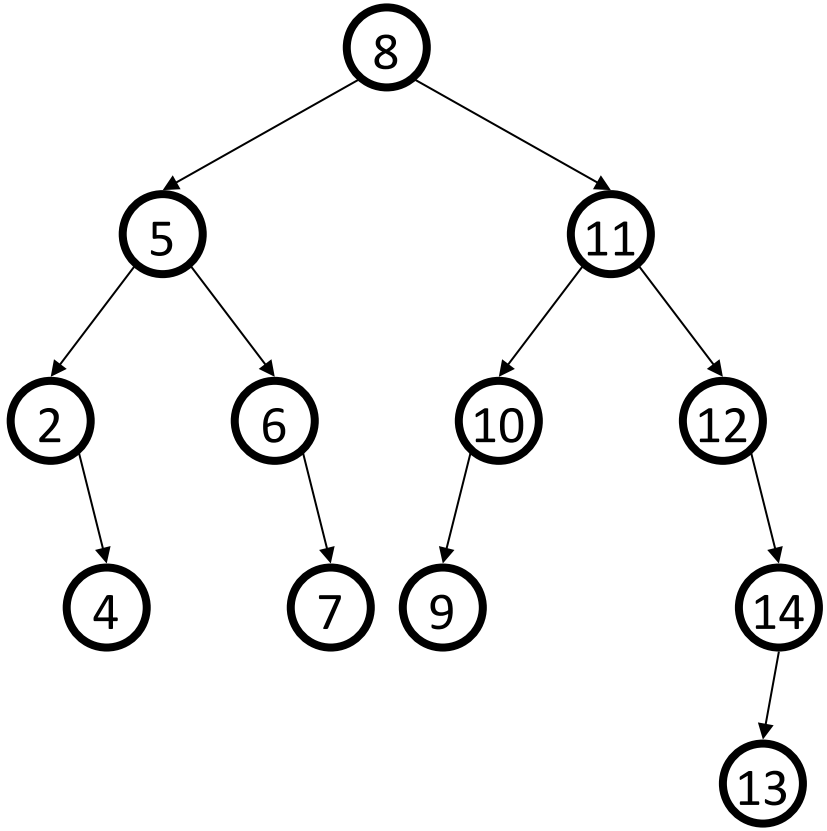
- Simpler to delete (no shifting). If element is re-added soon afterwards, simple to insert it again (no shifting)
- Can control removals and do them later in batches (amortized cost, we'll talk about this later today)

Minuses:

- Extra *space* for the "is-it-deleted" flag
- Data structure full of deleted nodes wastes *space*
- Now we can't use N in runtime: **`find`** $O($**`log`** $m)$ *time* where m is data-structure size (okay)

Review: Binary Search Tree

- Structure property ("binary")
  - Each node has ≤ 2 children
- Order property
  - All keys in left subtree smaller than node's key
  - All keys in right subtree larger than node's key
  - Result: easy to find any given key



BuildTree for BST

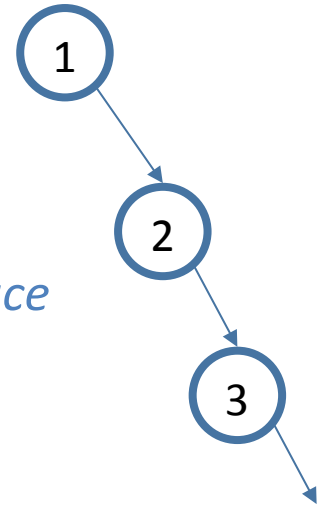
- Let's consider **`buildTree`** (Insert all, starting from an empty tree)

- Insert data 1, 2, 3, 4, 5, 6, 7, 8, 9 into an empty BST

  - If inserted in given order, what is the tree?

  - What big-O runtime for this kind of sorted input?

  - Is inserting in the reverse order any better?



$O(n^2)$
Not a happy place

Nope, the BST would just be the opposite

Intuition: Balanced BSTs are good

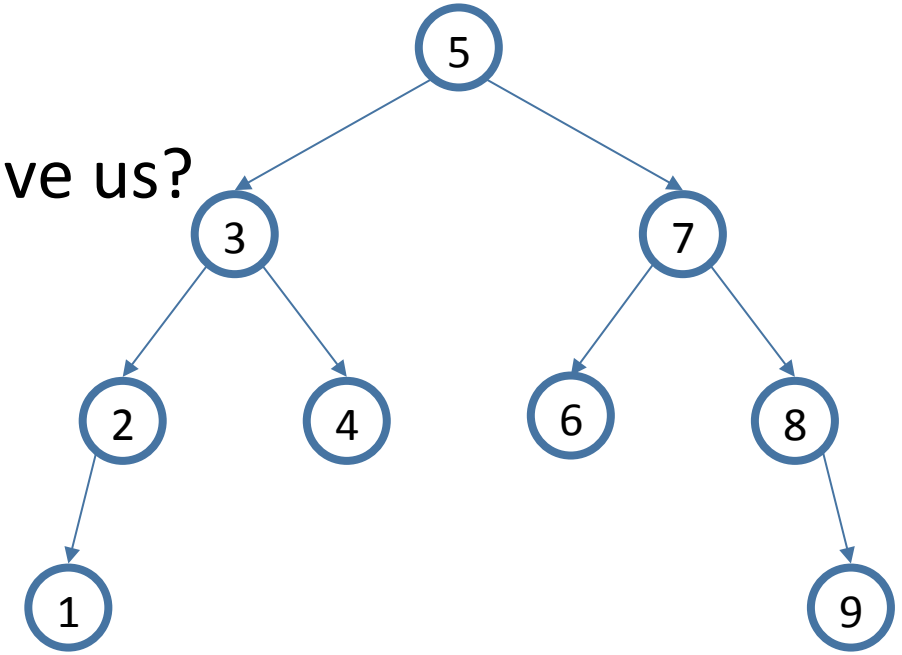
What if we re-arrange the data when inserting

- median first, then left median, right median, etc.
- 5, 3, 7, 2, 1, 4, 8, 6, 9

- What tree does that give us?

- What big-O runtime?

O(N log*N), definitely better*



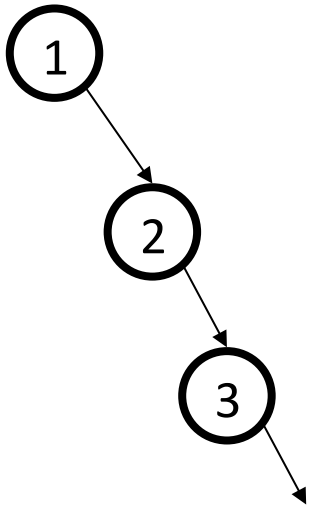
Intuition: Unbalanced BSTs are bad

- Even if we balance a BST during **`buildTree`**, a series of sorted data insertions can mess up our structure badly

- With a bad structure,

  all operations are *O(N):*

  - **`find`**
  - **`insert`**
  - **`delete`**



BST Operations Analysis

For BST with n nodes

	`insert`	`find`	`delete`
Worst Case (unbalanced)	*O(N)*	*O(N)*	*O(N)*
Average Case (balanced)	*O(logN)*	*O(logN)*	*O(logN)*

Hard to keep a BST balanced, so BSTs are only "probably" better as implementations than a sorted array. We'll see how to keep them balanced on Friday

Lazy Deletion for BSTs:

- Plusses:
  - Simpler: delete with **`findMin`** and **`findMax`** are difficult operations, this minimizes those traversals
  - Can do "real deletions" later as a batch
  - Some inserts can just "undelete" a tree node
- Minuses:
  - Can waste space and slow down find operations
  - Makes some operations more complicated with extra nodes in the tree

Keeping BSTs Balanced

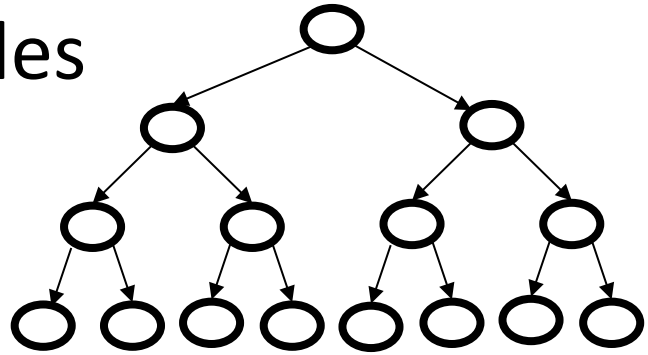
- For a BST with *N* nodes inserted in arbitrary order
  - Average height is $O(\log N)$ – intuition on Friday's slides, proof in text
  - Worst case height is $O(N)$
- Simple, commonly occurring cases, such as inserting in key order, lead to the worst-case scenario

Solution: Require a **Balance Condition** that maintains a nice structure:

1. Ensures depth is always $O(\log N)$ – strong enough!
2. Is efficient to maintain – not too strong!

Potential Balance Conditions

1. Left and right subtrees of every node have equal number of nodes



Too strong!
Only perfect trees ($2^n - 1$ nodes)

2. Left and right subtrees of every node have equal *height*

Too strong!
Only perfect trees ($2^n - 1$ nodes)

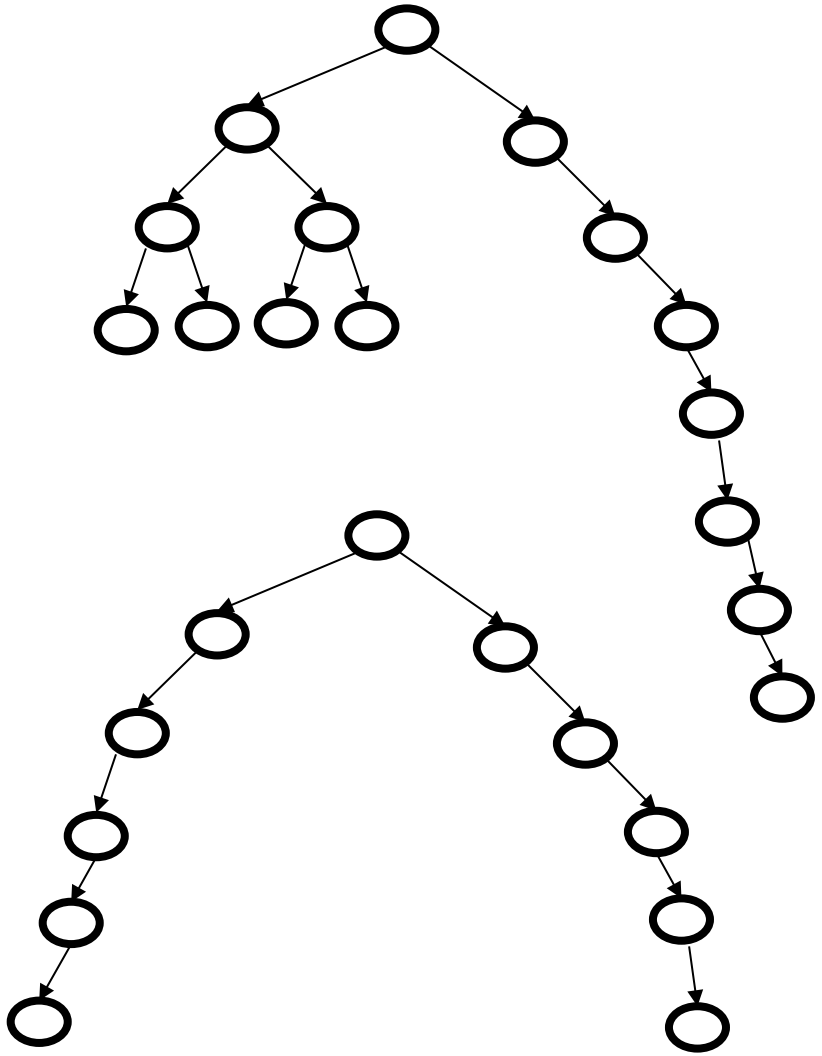
Potential Balance Conditions

3. Left and right subtrees of the *root* have equal number of nodes

Too weak!
Height mismatch example:

4. Left and right subtrees of the *root* have equal *height*

Too weak!
Double chain example:



The AVL Balance Condition

Left and right subtrees of *every node*
have *heights* **differing by at most 1**

Definition: **balance**(*node*) = height(*node*.left) – height(*node*.right)

AVL *property*: **for every node *x*, $-1 \leq \text{balance}(x) \leq 1$**

- Ensures small depth
  - Will prove this by showing that an AVL tree of height *h* must have a number of nodes *exponential* in *h*

- Efficient to maintain
  - Using single and double rotations

Any questions on Dictionaries or BSTs?

We'll explore the AVL balance condition and AVL trees more on Friday.

For now let's consider something we skipped when talking about asymptotic runtime analysis

Amortized Runtime Complexity

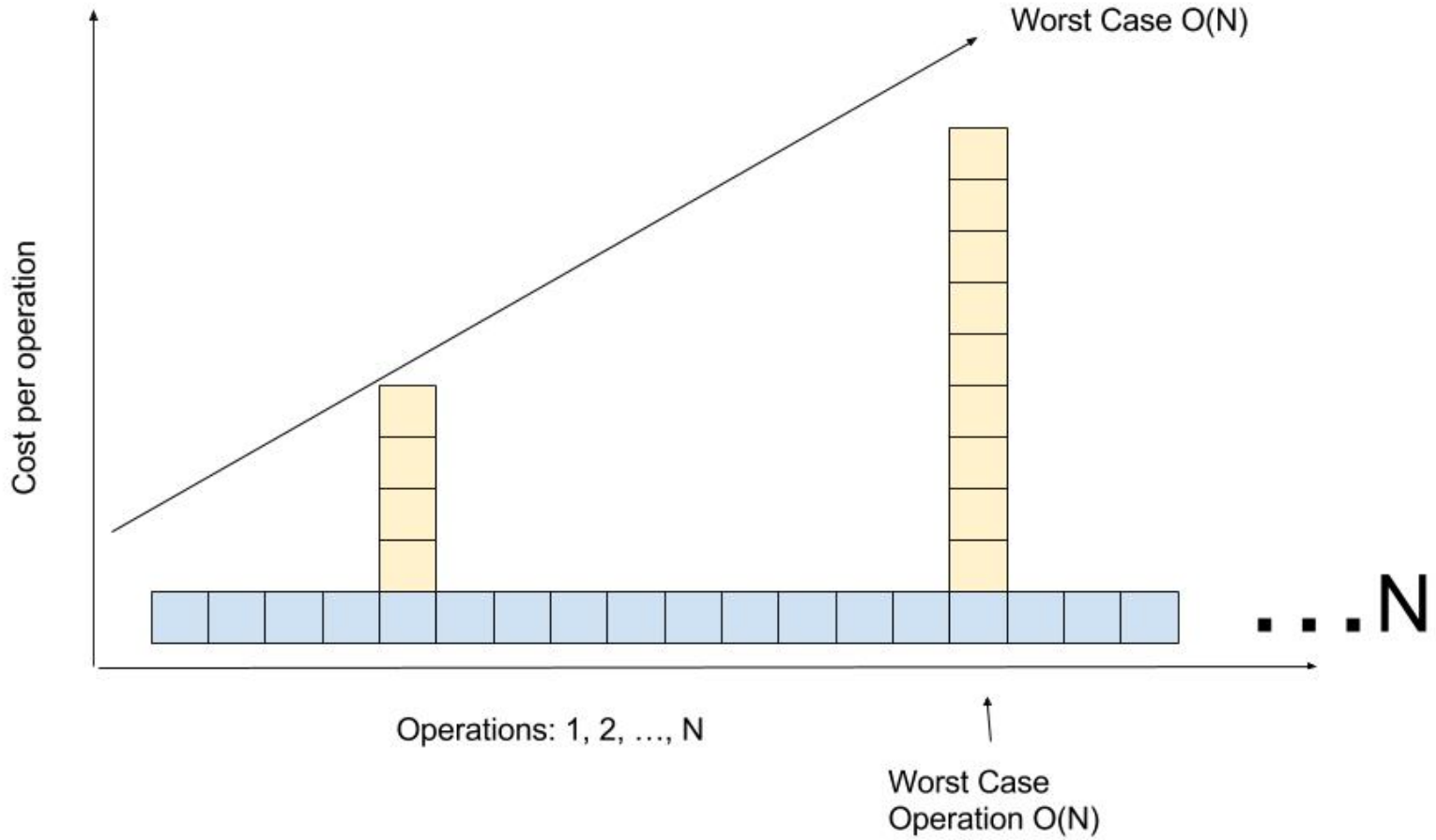
- Recall our plain-old stack implemented as an array that doubles its size if it runs out of room
  - How can we claim **`push`** is $O(\mathbf{1})$ time if resizing is $O(\mathbf{n})$ time?
  - We *can't*, but we *can* claim it's an $O(\mathbf{1})$ amortized operation
- What does amortized mean?
- When 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

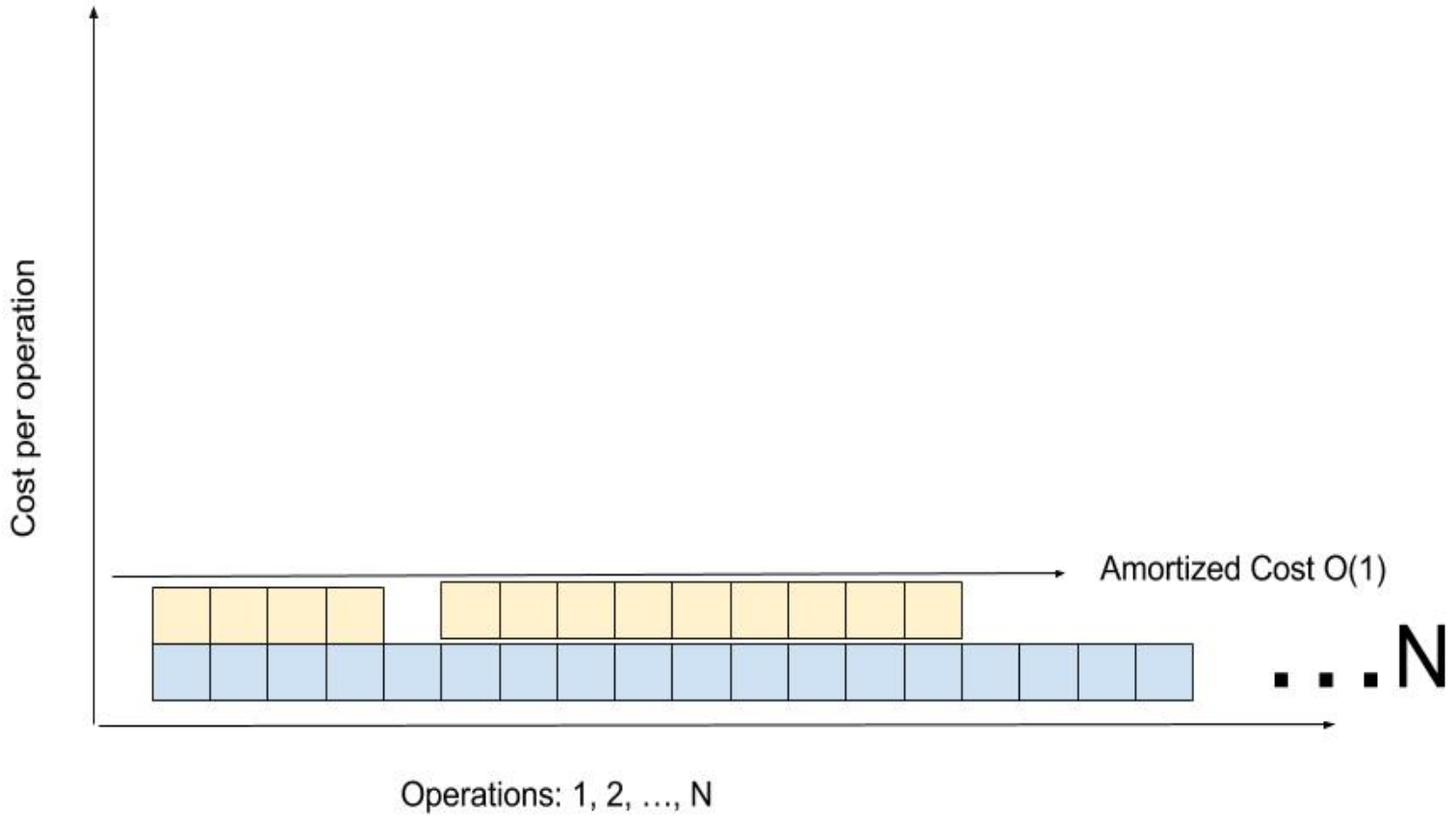
Amortized Runtime Intuition

Consider implementing a Stack with an Array. What if we had initially 5 empty slots, and every time it gets full, we add an additional size * 2 slots and have to copy over all the old data? What is the **worst case runtime for the `add(element)` operation?**



Amortized Runtime Intuition

Consider implementing a Stack with an Array. What if we had initially 5 empty slots, and every time it gets full, we add an additional size * 2 slots and have to copy over all the old data? What is the **amortized runtime for the `add(element)`** operation?



"Building Up Credit" Intuition

- 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

Amortized Runtime Complexity

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

Amortized bound: worst-case guarantee over sequences of operations

- Example: If any **n** operations take $O(\mathbf{n})$, then amortized $O(\mathbf{1})$
- 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 $O(\mathbf{f(n)})$

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 $O(1)$

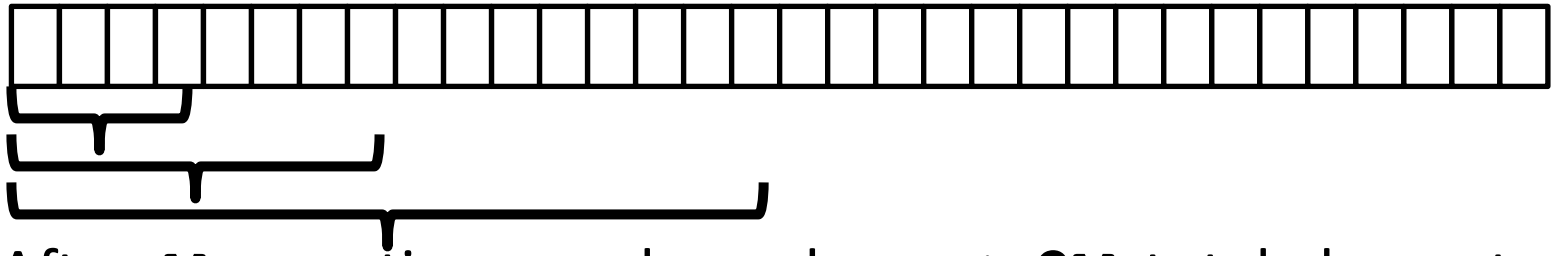
Need to show any sequence of **`M`** operations takes time $O(M)$

- 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 ≥ n/2
- So

  2M ≥ n > *number of element copies*

Other approaches

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

- If array doubles when full and 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, …

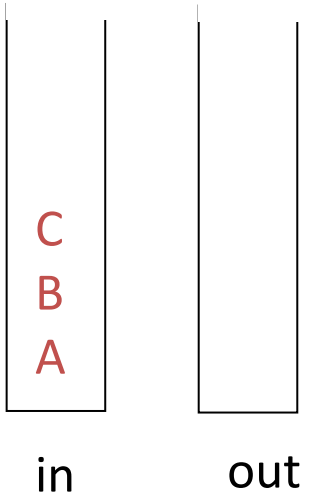
- If array doubles when full and shrinks when 3/4 empty, it is amortized $O(\mathbf{1})$
  - Proof is more complicated, but basic idea remains: by the time an expensive operation occurs, many cheap ones occurred

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();
  }
}
```

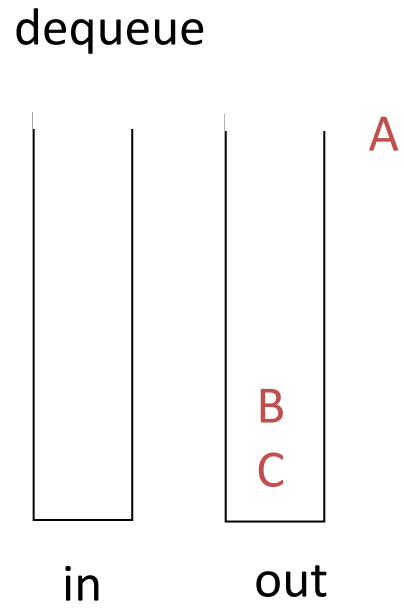
enqueue: A, B, C



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

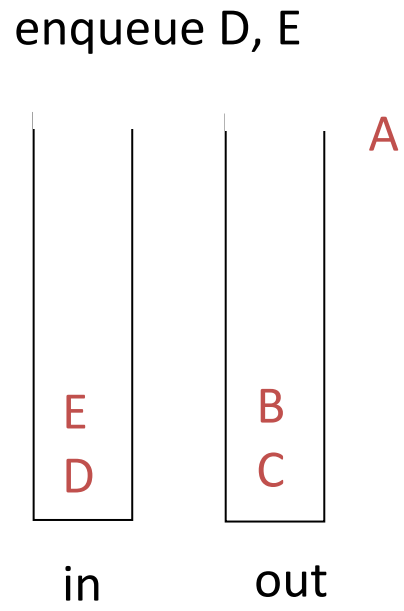
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

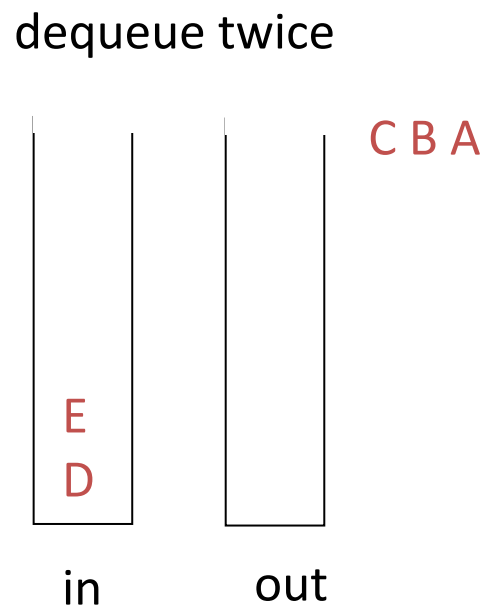
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



Example #2: Queue with two stacks

A clever and simple queue implementation using only stacks

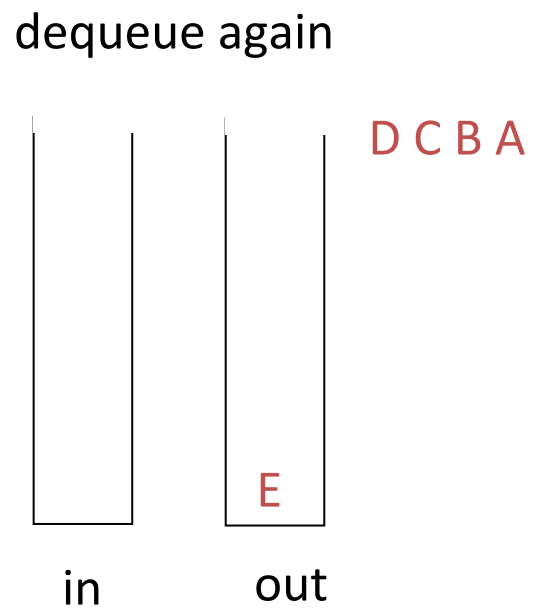
```
class Queue<E> {
  Stack<E> in  = new Stack<E>();
  Stack<E> out = new Stack<E>();
  void enqueue(E x){ in.push(x); }
  E dequeue(){
    if(out.isEmpty()) {
      while(!in.isEmpty()) {
        out.push(in.pop());
      }
    }
    return out.pop();
  }
}
```



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();
  }
}
```



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 $O(\texttt{log n})$ operations
  - 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