Lecture 2
ADTs, Stacks, Queues

Stacks

- The ***Stack*** ADT supports operations:
  - **`isEmpty`**: have there been same number of pops as pushes
  - **`push`**: takes an item
  - **`pop`**: raises an error if empty, else returns most-recently pushed item not yet returned by a pop
  - … (possibly more operations)

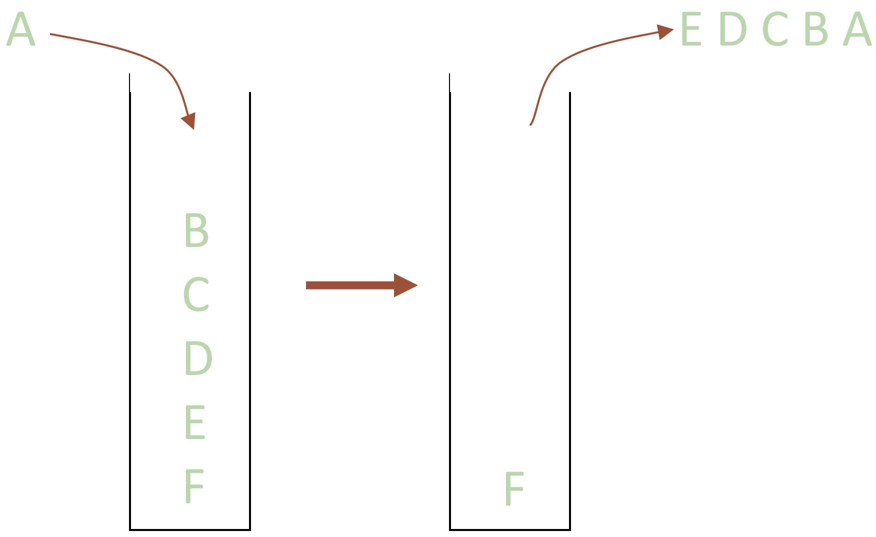
- A Stack data structure could use a linked-list or an array or something else, and associated algorithms for the operations

- One implementation is in the library **`java.util.Stack`**

The Stack ADT

Operations:

```
create
destroy
push
pop
top
is_empty
```



Can also be implemented with an array or a linked list

- The list in Python acts like a stack
- Type of elements is irrelevant

Why Stack ADT is useful

- It arises all the time in programming (e.g., see Weiss 3.6.3)
  - Recursive function calls
  - Balancing symbols (parentheses)
  - Evaluating postfix notation: 3 4 + 5 *
  - Clever: Infix ((3+4) * 5) to postfix conversion (see text)

- We can code up a reusable library

- We can communicate in high-level terms
  - “Use a stack and push numbers, popping for operators…”
  - Rather than, “create a linked list and add a node when…”

The Queue ADT

- Operations

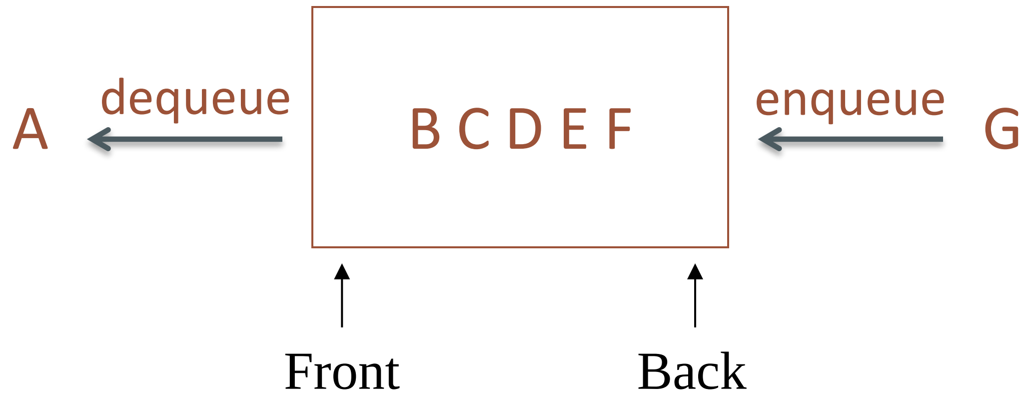
  `create`

  `destroy`

  `enqueue`

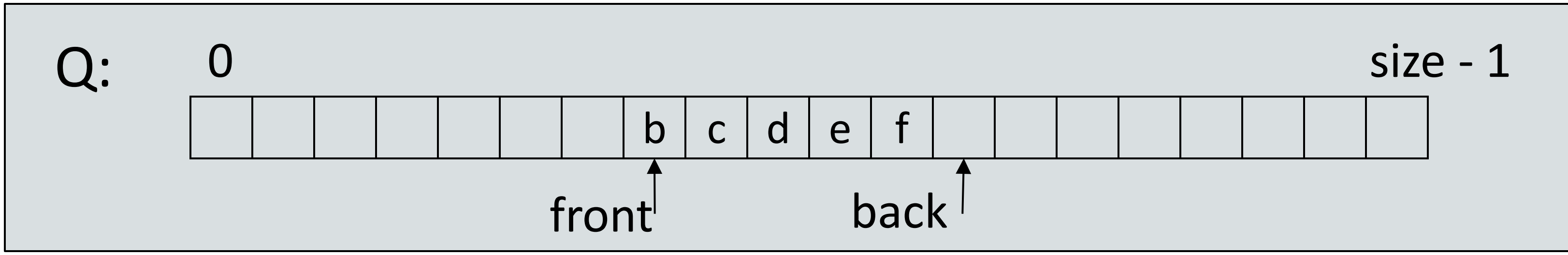
  `dequeue`

  `is_empty`



- Just like a stack except:
  - Stack: LIFO (last-in-first-out)
  - Queue: FIFO (first-in-first-out)
- Just as useful and ubiquitous

Circular Array Queue Data Structure



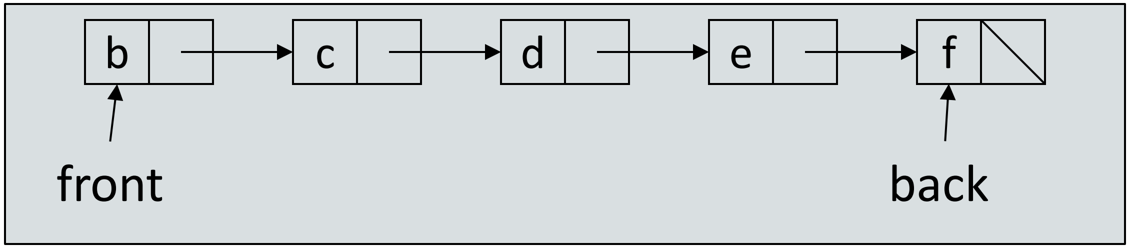

```
// Basic idea only!
enqueue(x) {
  Q[back] = x;
  back = (back + 1) % size
}
```

```
// Basic idea only!
dequeue() {
  x = Q[front];
  front = (front + 1) % size;
  return x;
}
```

Considerations:

- What if ***queue*** is empty?
  - Enqueue?
  - Dequeue?
- What if ***array*** is full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the k^{th} element in the queue?

Linked List Queue Data Structure




```
// Basic idea only!
enqueue(x) {
  back.next = new Node(x);
  back = back.next;
}
```

```
// Basic idea only!
dequeue() {
  x = front.item;
  front = front.next;
  return x;
}
```

Considerations:

- What if ***queue*** is empty?
  - Enqueue?
  - Dequeue?
- Can ***list*** be full?
- How to *test* for empty?
- What is the *complexity* of the operations?
- Can you find the k^{th} element in the queue?

Data Structure Analysis Practice

For each of the following, pick the best Data Structure (**Stack, Queue, either, or neither**) and Implementation (**Array, Linked List, either, or neither**):

- Maintain a collection of customers at a store with a relatively constant stream of customers at all times
- Keep track of a ToDo list
- Maintain a sorted student directory
- Manage the history of webpages visited to be used by the "back" button
- Store data and access the *kth* element often

Pseudocode

Describe an algorithm in the steps necessary, write the shape of the code but ignore specific syntax.

Algorithm: Count all elements in a list greater than x

Pseudocode:

```
int counter // keeps track of number > x
while list has more elements {
   increment counter if current element is > than x
   move to next element of list
}
```

Pseudocode Example 2

Algorithm: Given a list of names in the format “firstName lastName”, make a Map of all first names as keys with sets of last names as their values

Pseudocode:

```
create the empty result map
while list has more names to process {
    firstName is name split up until space
    lastName is name split from space to the end
    if firstName not in the map yet {
       put firstName in map as a key with an empty
       set as the value
    }
    add lastName to the set for the first name
    move to the next name in the list
}
```

Amortized Runtime Complexity

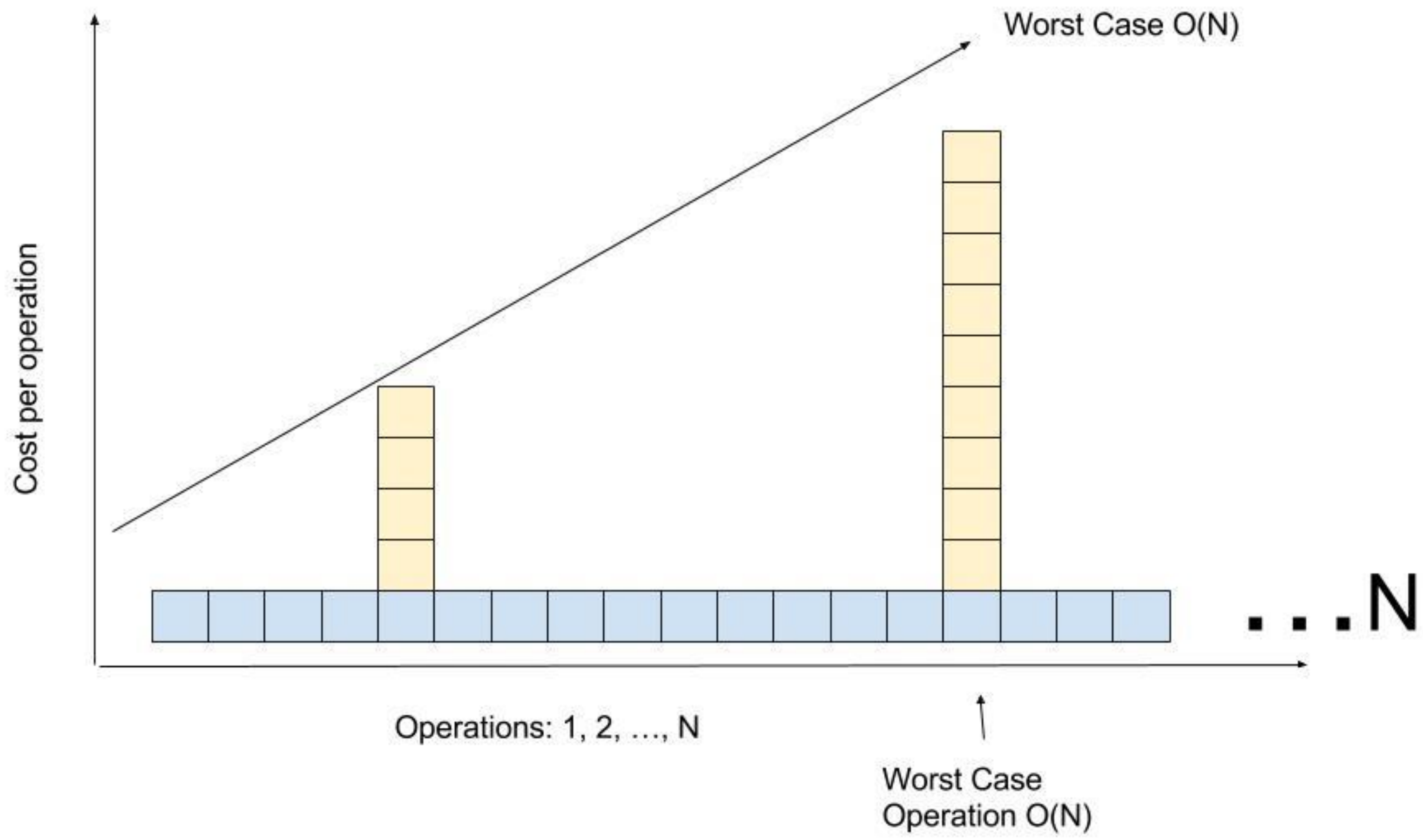
- Recall our plain-old stack implemented as an array that doubles its size if it runs out of room
  - How can we claim **`push`** is $O(\mathbf{1})$ time if resizing is $O(\mathbf{n})$ time?
  - We *can't*, but we *can* claim it's an $O(\mathbf{1})$ amortized operation
- What does amortized mean?
- When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

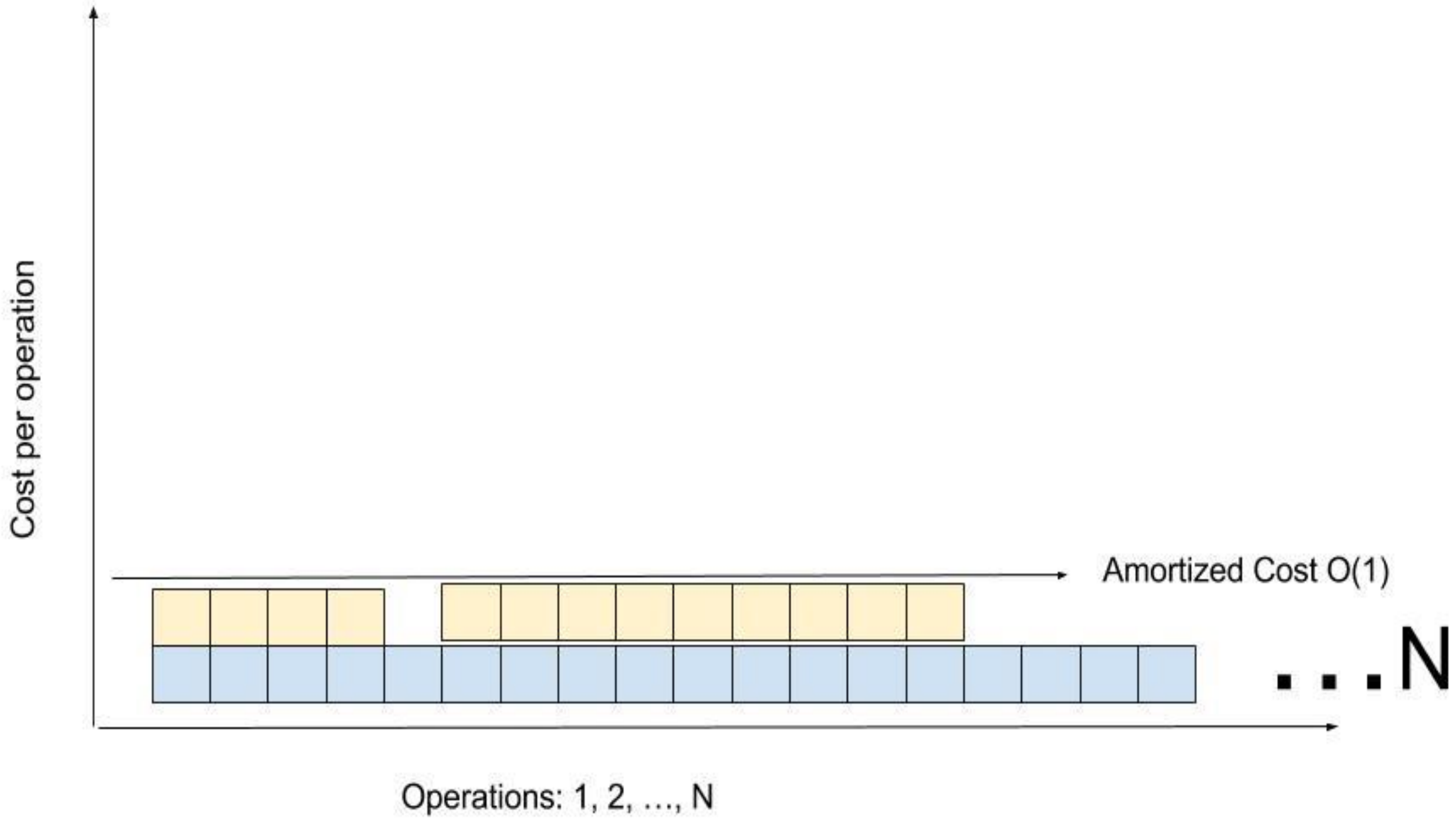
Amortized Runtime Intuition

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 $O(\mathbf{1})$

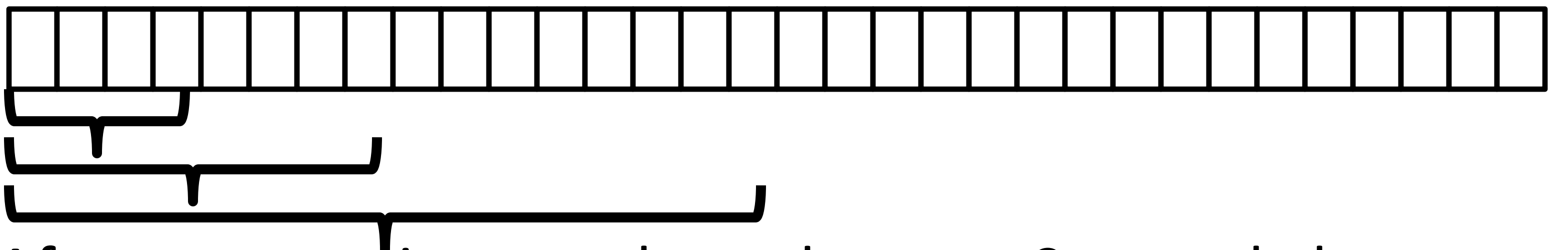
Need to show any sequence of **`M`** operations takes time $O(\mathbf{M})$

- Recall the non-resizing work is $O(\mathbf{M})$ (i.e., $\mathbf{M} * O(\mathbf{1})$)
- 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**:

  $$\mathbf{M} \geq \mathbf{n/2}$$
- So

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