**CIS 1210—Data Structures and Algorithms—Fall 2023**

**Stacks & Queues**—Tuesday, September 26 / Wednesday, September 27

# Readings

- Lecture Notes Chapter 13: Stacks & Queues

# Review: Stacks and Queues

An **abstract data type** (ADT) is an abstraction of a data structure; it specifies the type of data stored and the operations that can be performed, similar to a Java interface. Recall the Stack and Queue ADTs:

| **Stack** | **Queue** |
|---|---|
| • LIFO (Last-In-First-Out): the most recent element added to the stack will be removed first<br>• Supported operations:<br>– `push`: amortized $O(1)$<br>– `pop`: amortized $O(1)$<br>– `peek`: $O(1)$<br>– `isEmpty`: $O(1)$<br>– `size`: $O(1)$ | • FIFO (First-In-First-Out): the oldest/least recent element added to the queue will be removed first<br>• Supported operations:<br>– `enqueue`: amortized $O(1)$<br>– `dequeue`: amortized $O(1)$<br>– `peek`: $O(1)$<br>– `isEmpty`: $O(1)$<br>– `size`: $O(1)$ |

### Implementation Details

In this course, we implement stacks and queues using (dynamically resizing) arrays. In other words, we adjust the size of the array so that it is large enough to store all of its current elements but not large enough that it wastes space. The rules we will use for increasing or decreasing the size of a stack or queue's underlying array are as follows:

1. If the array of size $n$ is full, create a new array of size $2n$ and copy all elements into the new array.
2. If the array of size $n$ has less than $\frac{n}{4}$ elements in it, create a new array of size $\frac{n}{2}$ and copy all elements into the new array.

Note that we resize "down" when the array has $\frac{n}{4}$ elements in it (instead of when it has $\frac{n}{2}$ elements) to prevent "thrashing." If we resized "down" when the array has $\frac{n}{2}$ elements, consider the case where we `push` elements onto a stack until it resized "up." If we were to `pop` a single element, then we would have to resize "down," but then if we were to `push` another element, we would have to resize "up" again, so in the worst-case, every `push`/`pop` operation would require copying elements and creating new arrays, increasing our runtimes.

### Amortized Analysis

When calculating the runtimes of operations for stacks and queues, we perform amortized analysis. In amortized analysis, the amortized runtime of a single operation is equal to the time needed to perform a series of operations divided by the number of operations performed. For example, let $T(n)$ be the amount of time

needed to perform $n$ `push` operations. Then, the amortized runtime of a single `push` operation is equal to $\frac{T(n)}{n}$. Observe that we often perform amortized analysis in situations where the occasional operation takes much longer than the rest of the operations. Considering a stack, in the worst-case, a `push` operation takes $O(n)$ time because of array resizing, but otherwise most of the `push` operations take $O(1)$ time since we're just setting a value at an index of the array.

**Note:** Amortized analysis is **not** the same as average-case analysis, since it does not depend at all on the probability distribution of inputs. Instead, the total running time of a series of operations is bounded by the total runtime of the amortized operations.

# Problems

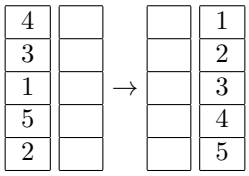
## Problem 1

You are given two stacks $S_1$ and $S_2$ of size $n$. Implement a queue using $S_1$, $S_2$, and a stack's `push`, `pop`, and/or `peek` methods. What are the (amortized) running times of your new `enqueue` and `dequeue` methods?

## Problem 2

You are given a full stack $S_1$ with distinct elements and an empty stack $S_2$, each of size $n$. Design an algorithm to sort the $n$ elements in increasing order from the top in $S_2$, using only $O(1)$ additional space beyond $S_1$ and $S_2$. What is the running time of your sorting algorithm?

**Example:**




*Hint:* Start with a smaller example: