**CIS 121—Data Structures and Algorithms with Java—Spring 2018**

**Stacks and Queues**—Monday, February 12 / Tuesday, February 13

# Learning Goals

During this lab, you will:

- Review stacks and queues.
- Learn amortized running time analysis and strengthen intuition for applying it to new problems.
- Practice using stacks and queues to accomplish a variety of tasks.

# Stacks and Queues

Recall the *stack* and *queue* ADTs (abstract data types) from lecture. Each is characterized by a specific way of removing elements and has a set of supported operations.

| **Stack** | **Queue** |
|---|---|
| • LIFO (last-in-first-out)—the most recent element that has been added to the stack will be removed first.<br>• Supported operations:<br>– `push`<br>– `pop`<br>– `peek`<br>– `isEmpty`<br>– `size` | • FIFO (first-in-first-out)—the least recent element that has been added to the queue will be removed first.<br>• Supported operations:<br>– `enqueue`<br>– `dequeue`<br>– `peek`<br>– `isEmpty`<br>– `size` |

## Implementation Details

Stacks and queues can be implemented "under the hood" with almost any data structure. In this course, we will implement stacks and queues using *expandable arrays.* 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 $\frac{n}{4}$ elements in it, create a new array of size $\frac{n}{2}$, and copy all elements into the new array.

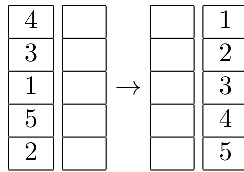
# Problems

## Problem 1: Sorting Using Stacks

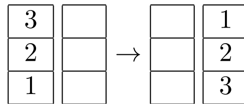
*Given:* A full stack $S_1$ of size $n$ and an empty stack $S_2$ of size $n$.

*Objective:* Sort the $n$ elements in ascending order in $S_2$. You may only use the given 2 stacks $S_1$ and $S_2$ (each of size $n$) and $O(1)$ additional space. What is the running time of your sorting procedure?

*Example:*




*Hint:* Start with a simpler example:




## Problem 2: Level-Order traversal of Binary Tree

*Given:* A binary tree of size n

*Objective:* Print out the level order traversal of the binary tree

*Example:* see below

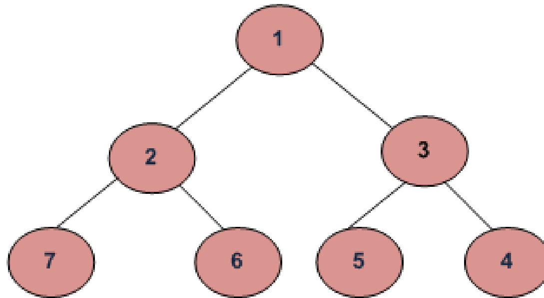



Figure 1: *For this tree, your function should print 1, 2, 3, 7, 6, 5, 4.*

## Problem 3: Spiral Order Tree Traversal

*Given:* A binary tree $T$.

*Objective:* Print the spiral order traversal of the tree $T$.

*Example:*

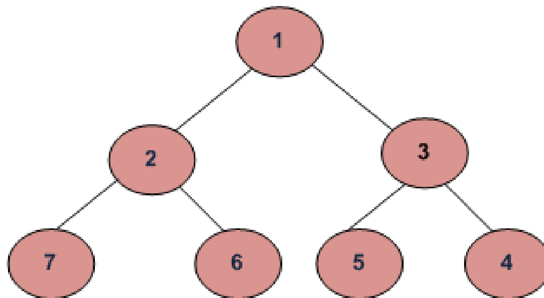



Figure 2: *For this tree, your function should print 1, 2, 3, 4, 5, 6, 7.*

*Hint:* Try using 2 stacks.

# Amortized Analysis

*Amortized analysis* refers to finding the time-averaged cost for a sequence of operations. In other words, it is the time required to perform a sequence of operations averaged over all the operations performed.

Since amortized analysis for the stack `push` operation was covered in lecture, we are going to take a closer look at the stack `pop` operation.

The worst case running time for a single `pop` operation is $O(n)$, since we may need to resize the array and copy the elements into it. Based on this running time, we might conclude that a tight bound for the worst case running time for $n$ `pop` operations is $O(n^2)$, since there are $n$ operations and each operation takes worst case $O(n)$ time; however, we can find a tighter bound through some careful analysis.

If we start from a full stack of size $n$, what is the total cost of a sequence of $n$ `pop` operations?

Initially, the array is of size $n$ and contains $n$ elements. To make our analysis simpler, let's immediately `pop` the first $\frac{n}{2}$ elements. Each of these `pop`s takes $O(1)$ time. Now our array is of size $n$ but contains only $\frac{n}{2}$ elements.

In accordance with our rules, we can `pop` $\frac{n}{4}$ more elements before resizing the array. Each of these `pop`s takes $O(1)$ time. Once we have `pop`'d those elements (leaving us with $\frac{n}{4}$ elements in our array), we must reduce the size of our array to $\frac{n}{2}$, and copy the remaining $\frac{n}{4}$ elements into the new array. Thus, the total cost for the first $\frac{3n}{4}$ `pop` operations is $T(\frac{3n}{4}) = \frac{n}{2} + \left(\frac{n}{4} + \frac{n}{2} + \frac{n}{4}\right)$.

We can apply identical analysis to the new array of size $\frac{n}{2}$ that contains $\frac{n}{4}$ elements. We get $\frac{1}{4}\left(\frac{n}{2}\right) = \frac{n}{8}$ `pop`s "for free", after which we resize the array to be of size $\frac{1}{2}\left(\frac{n}{2}\right) = \frac{n}{4}$ and copy the remaining $\frac{1}{4}\left(\frac{n}{2}\right) = \frac{n}{8}$ elements into the smaller array. Thus, the total cost for the first $\frac{7n}{8}$ `pop` operations is $T(\frac{7n}{8}) = \frac{n}{2} + \left(\frac{n}{4} + \frac{n}{2} + \frac{n}{4}\right) + \left(\frac{n}{8} + \frac{n}{4} + \frac{n}{8}\right)$.

Are you noticing a pattern?

Let's rewrite the expression slightly and continue to expand it:

$$T(n) = \frac{n}{2} + \left(\frac{1}{4}\left(\frac{n}{2^0}\right) + \frac{1}{2}\left(\frac{n}{2^0}\right) + \frac{1}{4}\left(\frac{n}{2^0}\right)\right) + \left(\frac{1}{4}\left(\frac{n}{2^1}\right) + \frac{1}{2}\left(\frac{n}{2^1}\right) + \frac{1}{4}\left(\frac{n}{2^1}\right)\right) + \\ + \left(\frac{1}{4}\left(\frac{n}{2^2}\right) + \frac{1}{2}\left(\frac{n}{2^2}\right) + \frac{1}{4}\left(\frac{n}{2^2}\right)\right) + \cdots + \left(\frac{1}{4}(4) + \frac{1}{2}(4) + \frac{1}{4}(4)\right)$$

We can now calculate the total cost of $n$ `pop` operations:

$$\begin{aligned} T(n) &\leq \frac{n}{2} + \sum_{i=0}^{\infty}\left(\frac{1}{4}\left(\frac{n}{2^i}\right) + \frac{1}{2}\left(\frac{n}{2^i}\right) + \frac{1}{4}\left(\frac{n}{2^i}\right)\right) \\ &= \frac{n}{2} + n\sum_{i=0}^{\infty}\frac{1}{2^i} \\ &= \frac{n}{2} + 2n \\ &\leq 3n \\ &= O(n) \end{aligned}$$

(The first term in the summation is the *cost of the initial* ***`pop`s***, the second term is the *cost of allocating* a new array, and the third term is the *cost of copying* the remaining elements into the new array.)

Thus, the *amortized* time complexity of a `pop` operation is $3 = O(1)$, even though the worst case time complexity of a single `pop` operation is $O(n)$.

## Problem 4: Queue With Two Stacks

*Given:* Two stacks $S_1$ and $S_2$, each of size $n$.

*Objective:* Implement a queue using $S_1$ and $S_2$. Your queue's `enqueue` and `dequeue` methods should be implemented using only your stacks' `push`, `pop`, and/or `peek` methods. What are the running times of your new queue's `enqueue` and `dequeue` methods?