Learning Goals

During this lab, you will:

- Review stacks and queues.
- Review 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.

<table>
<thead>
<tr>
<th>Stack</th>
<th>Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIFO (last-in-first-out)—the most recent element that has been added to the stack will be removed first.</td>
<td>FIFO (first-in-first-out)—the least recent element that has been added to the queue will be removed first.</td>
</tr>
<tr>
<td>Supported operations:</td>
<td>Supported operations:</td>
</tr>
<tr>
<td>push</td>
<td>enqueue</td>
</tr>
<tr>
<td>pop</td>
<td>dequeue</td>
</tr>
<tr>
<td>peek</td>
<td>peek</td>
</tr>
<tr>
<td>isEmpty</td>
<td>isEmpty</td>
</tr>
<tr>
<td>size</td>
<td>size</td>
</tr>
</tbody>
</table>

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.

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. 

---

1[^1]

[^1]: http://www.seas.upenn.edu/~cis121/current/lectures/stacksQueues.pdf
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.\footnote{The analysis for enqueue and dequeue is similar to that of push and pop, respectively.}

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 pops 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 pops takes $O(1)$ time. Once we have popped 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 $\frac{n}{8}$ elements into the new array. Thus, the total cost for the first $\frac{3n}{4}$ pops operations is $T\left(\frac{3n}{4}\right) = \frac{n}{2} + \left(\frac{n}{2} + \frac{n}{4} + \frac{n}{8}\right)$.

We can apply identical analysis to the new array of size $\frac{n}{2}$ that contains $\frac{n}{4}$ elements. We get $\frac{1}{4} \left(\frac{n}{2}\right) = \frac{n}{8}$ pops “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 $\frac{n}{8}$ elements into the smaller array. Thus, the total cost for the first $\frac{n}{8}$ pops operations is $T\left(\frac{n}{8}\right) = \frac{n}{2} + \left(\frac{n}{4} + \frac{n}{8} + \frac{n}{16}\right) + \left(\frac{n}{8} + \frac{n}{16} + \frac{n}{32}\right) + \cdots + \left(\frac{n}{4} + \frac{n}{8} + \frac{n}{16}\right)$.

Are you noticing a pattern?

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

$$T(n) = \frac{n}{2} + \left(\frac{1}{4} \left(\frac{n}{2^0}\right)\right) + \frac{1}{2} \left(\frac{n}{2^1}\right) + \frac{1}{4} \left(\frac{n}{2^2}\right) + \frac{1}{4} \left(\frac{n}{2^3}\right) + \frac{1}{2} \left(\frac{n}{2^4}\right) + \frac{1}{4} \left(\frac{n}{2^5}\right) + \cdots + \frac{1}{4} \left(\frac{n}{2^4}\right) + \frac{1}{2} \left(\frac{n}{2^5}\right) + \frac{1}{4} \left(\frac{n}{2^6}\right)$$

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

$$T(n) \leq \frac{n}{2} + \sum_{i=0}^{\infty} \left(\frac{1}{4} \left(\frac{n}{2^i}\right)\right)$$

$$= \frac{n}{2} + \sum_{i=0}^{\infty} \frac{1}{2^i}$$

$$= \frac{n}{2} + 2n$$

$$\leq 3n$$

$$= O(n)$$

(The first term in the summation is the cost of the initial pops, 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 $O(n)$.

**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:*
Problem 2: Spiral Order Tree Traversal

Given: A binary tree $T$.

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

Example:

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

Hint: Try using 2 stacks.

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

Problem 4: Generate Binary Numbers

Given: A number $n$ in base-10

Objective: Generate the binary representation (as a string) for all numbers from 1 up to $n$

Example:
• If \( n = 2 \), then we want to generate the numbers "1", "10"
• If \( n = 7 \), then we want to generate the numbers "1", "10", "11", "100", "101", "110", "111"

*Hint:* think about the last problem (level-order traversal) and how that could apply to this

**Problem 5: 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?