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.

<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.

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:*
Solution

To solve this problem, we will use the two given stacks, $S_1$ and $S_2$, and two extra variables $\text{max}$ and $\text{size}$.

Algorithm: Initialize $\text{max}$ to $-\infty$ and $\text{size}$ to 0.

1. pop all elements from $S_1$ and push them onto $S_2$. While pop’ing, keep track of the maximum element we have seen so far in $\text{max}$. Once we have push’ed all elements into $S_2$, the absolute maximum element will be stored in $\text{max}$.

2. pop all elements from $S_2$ and push all except the maximum element $\text{max}$ back into $S_1$.

3. push the maximum element (stored in $\text{max}$) into $S_2$. Now $S_1$ contains $n-1$ unsorted elements, and $S_2$ contains 1 sorted element.

4. Increment $\text{size}$ by 1. We will use $\text{size}$ to keep track of the number of sorted elements in $S_2$ so that we don’t pop them.

5. Repeat steps 1-4 until $\text{size} = n$. In Step 2, take care to only pop elements from $S_2$ until $S_2$ contains exactly $\text{size}$ elements. (The bottom $\text{size}$ elements in $S_2$ have already been sorted.)

When the procedure terminates, $S_1$ will be empty, and $S_2$ contains the elements in non-decreasing order.

Time complexity: The running time of our sorting procedure is $O(n^2)$, since for each element that we sort, we must push and pop at most $n$ elements.

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.
Solution

Algorithm: We use a queue to hold nodes that are to be visited. We first start with the queue containing the root node of the tree. While the queue is not empty, we dequeue an element from the queue, mark it as visited, and then enqueue its children into the queue.

- for the tree above, we first start with node 1 in the queue. We remove 1, mark it as visited, and add 2, 3 to the queue.
- We then remove 2 and 7, 6 to the queue. We remove 3 and add 5, 4 to the queue.
- Since all nodes in the queue at this point are leaves, we remove each node one by one until the queue is empty.

Problem 3: Spiral Order Tree Traversal

Given: A binary tree $T$.

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

Example:

![Tree Diagram]

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

Hint: Try using 2 stacks.

Solution

We will use two stacks, $S_1$ and $S_2$. We will use $S_1$ to hold elements in the same level that are being printed from left to right, and we will use $S_2$ to hold elements in the same level that are being printed from right to left. We observe that these stacks are disjoint (i.e., they contain no overlapping elements), and if a given node $n$ in $T$ is in $S_1$, then its two children should be in $S_2$ (and vice versa).

Algorithm: First, push the root of the tree $T$ onto stack $S_2$. The following procedure will loop until both $S_1$ and $S_2$ are empty.

- While $S_2$ is not empty, pop the top element $n$ from $S_2$. Print $n$. If $n$ has a right child, push it onto the other stack $S_1$. Then, if $n$ has a left child, push it onto $S_1$. Continue this step until $S_2$ is empty.
- While $S_1$ is not empty, pop the top element $n$ from $S_1$. Print $n$. If $n$ has a left child, push it onto the other stack $S_2$. Then, if $n$ has a right child, push it onto $S_2$. Continue this step until $S_1$ is empty.

Time and space complexity: If the tree $T$ contains $n$ nodes, this solution takes $O(n)$ time and $O(n)$ extra space.
Alternate solution (slower)

printSpiral(tree):
    leftToRight <- True
    for d = 1 to tree.height():
        printGivenLevel(tree, d, leftToRight)
        leftToRight <- !leftToRight

printGivenLevel(tree, level, leftToRight):
    if level is 1:
        print tree.data()
    else:
        if leftToRight == true:
            printGivenLevel(tree.left(), level - 1, leftToRight)
            printGivenLevel(tree.right(), level - 1, leftToRight)
        else:
            printGivenLevel(tree.right(), level - 1, leftToRight)
            printGivenLevel(tree.left(), level - 1, leftToRight)

Time and space complexity: If the tree \( T \) is balanced and has a height of \( O(\lg n) \), then this solution has a running time of \( O(n \lg n) \); however, if the tree \( T \) is unbalanced (e.g., if \( T \) actually resembles a linked list), then this solution takes \( O(n^2) \) time. Since the recursion can nest up to the height of the tree, this solution takes \( O(h) \) extra space where \( h \) is the height of the tree.

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

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.²

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 \( \frac{n}{4} \) elements (leaving us with \( n \) elements in our array), we must reduce the size of our array to \( \frac{n}{2} \), and copy the remaining \( n \) elements into the new array. Thus, the total cost for the first \( \frac{3n}{4} \) pop operations is \( T\left(\frac{3n}{4}\right) = \frac{n}{2} + \left(\frac{n}{2} + \frac{n}{2} + \frac{n}{4}\right) \).

We can apply identical analysis to the new array of size \( \frac{n}{2} \) that contains \( n \) 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 \( \frac{n}{4} \) elements into the smaller array. Thus, the total cost for the first \( \frac{7n}{8} \) pop operations is \( T\left(\frac{7n}{8}\right) = \frac{n}{2} + \left(\frac{n}{4} + \frac{n}{4} + \frac{n}{4}\right) + \left(\frac{n}{8} + \frac{n}{4} + \frac{n}{4}\right) \).

Are you noticing a pattern?

¹http://www.seas.upenn.edu/~cis121/current/lectures/stacksQueues.pdf
²The analysis for enqueue and dequeue is similar to that of push and pop, respectively.
Let’s rewrite the expression slightly and continue to expand it:

\[ T(n) = \frac{n}{2} + \left( \frac{1}{4} \left(\frac{n}{2^0}\right) + \frac{1}{2} \left(\frac{n}{2^1}\right) + \frac{1}{4} \left(\frac{n}{2^2}\right) \right) + \left( \frac{1}{4} \left(\frac{n}{2^1}\right) + \frac{1}{2} \left(\frac{n}{2^2}\right) + \frac{1}{4} \left(\frac{n}{2^3}\right) \right) + \cdots \]

We can now calculate the total cost of \( n \) \texttt{pop} operations:

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

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

\[ = \frac{n}{2} + 2n \]

\[ \leq 3n \]

\[ = O(n) \]

(The first term in the summation is the \textit{cost of the initial} \texttt{pop}s, the second term is the \textit{cost of allocating} a new array, and the third term is the \textit{cost of copying} the remaining elements into the new array.)

Thus, the \textit{amortized} time complexity of a \texttt{pop} operation is \( 3 = O(1) \), even though the worst case time complexity of a single \texttt{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 \texttt{enqueue} and \texttt{dequeue} methods should be implemented using only your stacks’ \texttt{push}, \texttt{pop}, and/or \texttt{peek} methods. What are the running times of your new queue’s \texttt{enqueue} and \texttt{dequeue} methods?

**Solution**

The solution for Problem 4 is very similar to that of Problem 3.

- **\texttt{enqueue}(x):**
  1. \texttt{push} \( x \) into \( S_1 \).

- **\texttt{dequeue}:**
  1. If \( S_2 \) is empty, \texttt{pop} all elements from \( S_1 \) and \texttt{push} them into \( S_2 \).
  2. If \( S_2 \) is still empty, return \texttt{Nil}.
  3. Else \texttt{pop} an element from \( S_2 \) and return it.

*Time complexity:* The running time of \texttt{enqueue}(x) is clearly \( O(1) \). The running time for \texttt{dequeue} is a bit trickier. If we consider that each element will be in each Stack exactly once, then we realize that each element will be pushed exactly twice and popped exactly twice. Thus, the amortized running time of \texttt{dequeue} is \( O(1) \).