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. • Supported operations: – `push` – `pop` – `peek` – `isEmpty` – `size`	• FIFO (first-in-first-out)—the least recent element that has been added to the queue will be removed first. • Supported operations: – `enqueue` – `dequeue` – `peek` – `isEmpty` – `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.

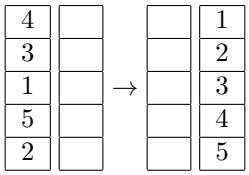
Problems

Problem 1: Sorting Using Stacks

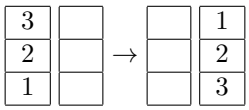
Given: A full stack S_1 of size n and an empty stack S_2 of size n.

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:



Solution

To solve this problem, we will use the two given stacks, S_1 and S_2, and two extra variables `max` and `size`.

Algorithm: Initialize `max` to $-\infty$ and `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 `max`. Once we have `push`'ed all elements into S_2, the absolute maximum element will be stored in `max`.
2. `pop` all elements from S_2 and `push` all except the maximum element `max` back into S_1.
3. `push` the maximum element (stored in `max`) into S_2. Now S_1 contains $n-1$ unsorted elements, and S_2 contains 1 sorted element.
4. Increment `size` by 1. We will use `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 `size` $= n$. In Step 2, take care to only `pop` elements from S_2 until S_2 contains exactly `size` elements. (The bottom `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

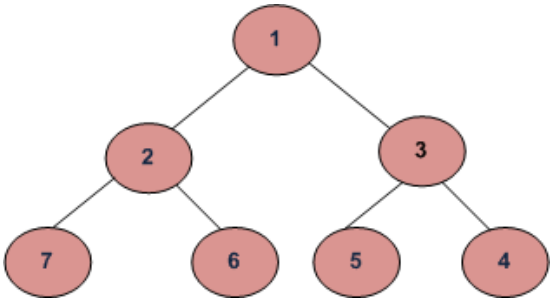


Figure 1: *For this tree, your function should print 1, 2, 3, 7, 6, 5, 4.*

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:

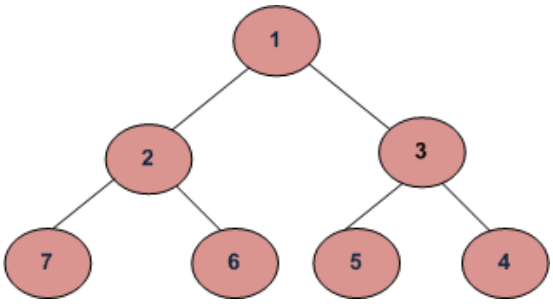


Figure 2: *For this tree, your function should print 1, 2, 3, 4, 5, 6, 7.*

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.

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?

Solution

`enqueue(x)`:

1. `push x` into S_1.

`dequeue`:

1. If S_2 is empty, `pop` all elements from S_1 and `push` them into S_2.
2. If S_2 is still empty, return NIL.
3. Else `pop` an element from S_2 and return it.

Time complexity: The running time of `enqueue(x)` is clearly $O(1)$. The running time for `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 `dequeue` is $O(1)$.