Portfolio Milestone I

Christopher Rodriguez

Colorado State University Global

WC21-CSC200-2: Computer Science Fundamentals

Prof. Chintan Thakkar

2022-02-06

Portfolio Milestone I

This is the first portfolio milestone for my class, where I will demonstrate knowledge of stacks, queues, and basic algorithm design.

I have chosen to use *Guile Scheme* as my form of (pseudo)code, as I am familiar with it and it is quite brief. Using an executable language as one's pseudocode allows for a confirmation of whether specific program flows will work.

In order to keep things brief, I've also used global variables and mutability to simplify the assignment and comparison functions. This would be avoided if this were not a pseudocode-level work: The important thing here was to show the algorithms and the program flow, not the overall structure.

And in order to quell confusion, I am using the queue library provided by Guile as my stack and queue objects, rather than implementing my own. For the stack-based operations, I have ensured a LIFO operative restriction remains throughout.

There are three tasks put forth for this milestone:

1. Write an algorithm that sets bottom equal to the last element in the stack, leaving the stack unchanged.
2. Write an algorithm to create a copy of myStack, leaving myStack unchanged.
3. Write an algorithm that sets last equal to the last element in a queue, leaving the queue empty.

Last Element of Stack

There are two ways I can concieve to complete this task.

First, there is the iterative approach: create a temporary, empty stack. Push each member to it as we pop it off of the original stack. Once the original stack is empty, pop the first item off of the temporary stack and set last equal to it before then pushing it back onto the original stack. Then pop all of the remaining members off of the temporary stack, pushing them back onto the original stack.

My preference, however, would be to use recursion: Basically, we will make an algorithm that checks to see if the stack only has one member. If so, this is our value, so we set last to it and do not recurse. If not, we pop the first member off of the stack, and recurse. Once we are past recursion, we then push the popped member *in each recursion* back onto the stack, which will naturally fall into preserving the order.

Here is the relevant code for the algorithm outlined above:

```
(define (last-element-of-stack stack)
          (let ((curr-value (q-pop! stack)))
            (if (q-empty? stack)
                (set! *last* curr-value)
                (last-element-of-stack stack))
            (q-push! stack curr-value)))
```

Copy Stack

This task is largely implementable using a modified version of the above function, where we instead push each value to two stacks instead of one.

Because of this change, we can now use a different flow-control mechanism---the *unless* clause---to get this work done. This keeps our lines of code down, making the concept as easy to understand as it originally was.

```
(define (copy-stack original new)
          (let ((curr-value (q-pop! original)))
            (unless (q-empty? original)
              (copy-stack original new))
            (q-push! original curr-value)
            (q-push! new curr-value)))
```

Both of these functions rely on a language with (and thus, on the concept of) tail call optimization. This turns an otherwise expensive amount of recursion into a series of jump statements, and lets us use recursion to avoid unnecessary structures like duplicate stacks and iterative bodies.

Last Element of Queue (And Empty)

There are a few ways we could execute this assignment: we could simply reference the last element of the queue with a tail (or rear) call, and then empty the queue in a single command by deleting it. Or, we could iterate through the queue and set last to each member, stopping once the queue is empty.

I have chosen the second method, though the first might be more time efficient. Here is the relevant section of code:

```
(while
  (not
    (q-empty? queue))
  (set! *last*
    (deq! queue)))
```

The basic algorithm would proceed as outlined above: while the queue is not empty, dequeue the first member of the queue, setting the value last equal to its value. If the queue is empty, we are finished, and the last value is set properly to the last member of the queue.

Here is a full implementation, with comments:

```
(use-modules (ice-9 q)) ;; Queue Library
(define queue (make-q)) ;; Make our queue
(define *last* #f)      ;; Define global variable "last"
```

```
  (do ((i 0 (1+ i)))       ;; Prepopulate queue with 0..4, five values
      ((> i 4))
    (enq! queue2 i))       ;; Prepopulate the queue with test values.
;; Setup Complete. Actual algorithm below; executes as follows:
;;
;; So long as the queue is not empty, we set last to the first item
;; in the queue, removing it from the queue as we do so (dequeuing it).
;; Once the queue is empty, return the empty queue (which will be a
;; falsey value (#f)).
  (while (not (q-empty? queue))
    (set! *last* (deq! queue)))
```

Conclusion

These three tasks, while very similar, require different levels of thought and a firm understanding of both stack and queue data structures. Though these are not production-ready algorithms, I feel as though they demonstrate a grasp of these concepts as well as the ability to take arbitrary similar problems and to produce adequate solutions.