**Mason Shepherd**

**Homework 4: due February 13th 11:59PM.**

The following Exercises are from Chapter 6 of the textbook *Data Structures and Algorithms in Java, 6th edition, Wiley, 2014.*

R-6.13 Suppose you have a deque *D* containing the numbers (1,2,3,4,5,6,7,8), in this order. Suppose further that you have an initially empty queue *Q*. Give a code fragment that uses only *D* and *Q* (and no other variables) and results in *D* storing the elements in the order (1, 2, 3, 5, 4, 6, 7, 8).

Ans:

Question is invalid. *D* already stores the elements (1,2,3,4,5,6,7,8) in that order. (???)

…if we wanted to store these elements in *D* in the reverse of its original order:

public void revDeqWithQ(Deque D, Queue Q){ // *D* is nonempty, *Q* is empty

while (!D.isEmpty()){

Q.enqueue(D.removeLast())

D.size()--

}

while (!Q.isEmpty()){

D.addLast(Q.dequeue())

Q.size()--

}

}

C-6.19 ***Postfixnotation*** is an unambiguous way of writing an arithmetic expression without parentheses. It is defined so that if “(*exp*1)**op**(*exp*2)” is a normal fully parenthesized expression whose operation is **op**, the postfix version of this is “*pexp*1 *pexp*2 **op**”, where *pexp*1 is the postfix version of *exp*1 and *pexp*2 is the postfix version of *exp*2. The postfix version of a single number or variable is just that number or variable. So, for example, the postfix version of “((5 + 2) ∗ (8 − 3))/4” is “5 2 + 8 3 − ∗ 4 /”. Give a pseudocode description of how to evaluate an expression in postfix notation.

Ans:

{on next page for clarity}

**Algorithm** evalPostfix(*A*): {where *A* is some data structure representing the postfix}

numbers = new stack

ops = new stack

i = {first element in the structure}

**while** A.size() != 0, **do**

**if** i == a number, **then**

numbers.push(i)

i = {next element}

A.size()--

**else**

exp1 = numbers.pop()

exp2 = numbers.pop()

**if** exp2 == NULL, **then**

**return** “Error: invalid postfix notation.”

k = (exp1) i (exp2) {computation of *exp1* and *exp2* by operation *i*}

numbers.push(k)

i = {next element}

A.size()--

**return** numbers.pop()

C-6.32 Suppose you have two nonempty stacks *S* and *T* and a deque *D*. Give a pseudocode description how to use *D* so that *S* stores all the elements of *T* below all of its original elements, with both sets of elements still in their original order.

Ans:

**Algorithm** buryStackUnderStack(*T*, *S*, *D*): {stack *T* will go under stack *S* via deque *D*}

**while** S.size != 0 **do**

D.addFirst(S.pop())

S.size()-- {if this isn’t done automatically with pop() method}

**while** T.size() != 0 **do**

D.addFirst(T.pop())

T.size()-- {^same note}

**while** D.size != 0 **do**

S.push(D.removeFirst())

D.size()-- {^}

P-6.38 The introduction of Section6.1 notes that stacks are often used to provide “undo” support in applications like a Web browser or text editor. While support for undo can be implemented with an unbounded stack, many applications provide only ***limited*** support for such an undo history, with a fixed-capacity stack. When push is invoked with the stack at full capacity, rather than throwing an exception, a more typical semantic is to accept the pushed element at the top while “leaking” the oldest element from the bottom of the stack to make room. Describe how to implement such a LeakyStack abstraction, using a circular array (no code is needed).

Ans:

Using a circular array in a stack’s LIFO format…

*f* would now represent the “bottom” element in the stack and would be “leaked” when the array is full and a new element is pushed onto the stack. It would be updated with every “leak” by *f* += 1 or *f* = 0 if it had previously been the last index in the array.

If the array is not full, the formula for the next open position:

*openIndex* = (*f* + *size*) % *length*,

would be used to find the position for the push() method.

However, if the array was at full capacity, after *f* is “leaked” and updated to *f* + 1, the next open position would then be *f* – 1.

For the pop() method, if the array was not full, it would pop the element at the index found by: *popIndex* = ((*f* + *size*) % *length*) – 1.

If the array was full, then the index could more easily be found by *f* – 1.