CSE143X Section #5 Problems

Queues should be constructed using the Queue<E> interface and the LinkedList<E> implementation:

```
Queue<String> q = new LinkedList<>();
```

Stacks should be constructed using the Stack<E> class (there is no interface):

```
Stack<String> s = new Stack<>();
```

For Stack<E>, you are limited to the following operations:

```
public void push(E value)  // push given value onto top of the stack
public E pop()             // removes and returns the top of the stack
public boolean isEmpty()   // returns whether or not stack is empty
public int size()          // returns number of elements in the stack
```

For Queue<E> you are limited to the following operations:

```
public void add(E value)  // inserts given value at the end of the queue
public E remove()         // removes and returns the front of the queue
public boolean isEmpty()  // returns whether or not queue is empty
public int size()         // returns number of elements in the queue
```

Each problem will indicate what kind of structure you can use as auxiliary storage. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You should not use a foreach loop or iterator in solving these problems.

1. Write a method splitStack that takes a stack containing a list of integers and that splits it into negatives and nonnegatives. The numbers in the stack should be rearranged so that all the negatives appear on the bottom of the stack and all the nonnegatives appear on the top of the stack. In other words, if after this method is called you were to pop numbers off the stack, you would first get all the nonnegative numbers (at the top) and then get all the negative numbers (at the bottom).

   It does not matter what order the numbers appear in as long as all the negatives appear lower in the stack than all the nonnegatives. You should use a single queue as auxiliary storage to solve this problem. Method splitStack should take a single parameter: the stack to be split.

2. Write a method stutter that takes a stack containing a list of integers and that replaces every value in the stack with 2 of that value. For example, suppose the stack stores these values:

   ```
   bottom [3, 7, 1, 14, 9] top
   ```

   Then the stack should store these values after the method terminates:

   ```
   bottom [3, 3, 7, 7, 1, 1, 14, 14, 9, 9] top
   ```

   Notice that you must preserve the original order. In the original list the 9 was at the top and would have been popped first. In the new stack the two 9's would be the first values popped from the stack.

   You should use a single queue as auxiliary storage to solve this problem. Method stutter should take a single parameter: the stack containing the list of integers to be stuttered.

3. Write a method equals that takes as parameters two stacks containing lists of integers and that returns true if the two stacks are equal and that returns false otherwise. Two stacks are considered equal if they store the same number of values and those values appear in the same order. Your method is to examine the two stacks but must return them to their original state before terminating. For example, if the given these stack variables:

```
s1          bottom [3, 18, 9, 42] top
s2          bottom [3, 18, 9, 42] top
s3          bottom [3, 18, 9, 42, 17] top
s4          bottom [42, 9, 18, 3] top
```

the call equals(s1, s2) would return true and the calls equals(s1, s3), equals(s1, s4), equals(s2, s3), and equals(s3, s4) would return false. Two empty stacks would be considered equal to each other.

You are to use one stack as auxiliary storage to solve this problem. Your method should be efficient by returning false immediately if the two stacks are not of the same size and by not examining more values from the two stacks once it has found a difference.

4. Write a method reverseHalf that reverses the order of half the elements of a queue of integers. Your method should reverse the order of all the elements in odd-numbered positions (position 1, 3, 5, etc) assuming that the first value in the queue has position 0.

For example, if the queue originally stores this sequence of integers when the method is called:

```
front [1, 8, 7, 2, 9, 18, 12, 0] back
```

it should store the following values after the method finishes executing:

```
front [1, 0, 7, 18, 9, 2, 12, 8] back
```

Notice that numbers in even positions (positions 0, 2, 4, 6) have not moved. That subsequence of integers is still:

```
(1, 7, 9, 12)
```

But notice that the numbers in odd positions (positions 1, 3, 5, 7) are now in reverse order relative to the original. In other words, the original subsequence:

```
(8, 2, 18, 0)
```

has become:

```
(0, 18, 2, 8)
```

You should use a single stack as auxiliary storage to solve this problem. Method reverseHalf should take a single parameter: the queue to modify.

5. Write a method isPalindrome that takes a Queue of integers as a parameter and that returns whether or not the numbers in the queue represent a palindrome (true if they do, false otherwise). A sequence of numbers is considered a palindrome if it is the same in reverse order. For example, suppose a Queue called q stores this sequence of values:

```
front [3, 8, 17, 9, 17, 8, 3] back
```

Then the following call:

```
isPalindrome(q)
```

should return true because this sequence is the same in reverse order. If the list had instead stored these values:

```
front [3, 8, 17, 9, 4, 17, 8, 3] back
```

the call on isPalindrome would instead return false because this sequence is not the same in reverse order (the 9 and 4 in the middle don't match).

The empty queue should be considered a palindrome. You may not make any assumptions about how many elements are in the queue and your method must restore the queue so that it stores the same sequence of values after the call as it did before.

You are to use one stack as auxiliary storage to solve this problem.

6. Write a method isConsecutive that takes a stack of integers as a parameter and that returns whether or not the stack contains a sequence of consecutive integers starting from the bottom of the stack (returning true if it does, returning false if it does not).

Consecutive integers are integers that come one after the other, as in 5, 6, 7, 8, 9, etc. So if a stack s stores the following values:

```
bottom [3, 4, 5, 6, 7, 8, 9, 10] top
```

then the call:

```
isConsecutive(s)
```

should return true. If the stack had instead contained this set of values:

```
bottom [3, 4, 5, 6, 7, 8, 9, 10, 12] top
```

then the call should return false because the numbers 10 and 12 are not consecutive. Notice that we look at the numbers starting at the bottom of the stack. The following sequence of values would be consecutive except for the fact that it appears in reverse order, so the method would return false:

```
bottom [3, 2, 1] top
```

Your method must restore the stack so that it stores the same sequence of values after the call as it did before. Any stack with fewer than two values should be considered to be a list of consecutive integers.

You are to use one queue as auxiliary storage to solve this problem.

7. Write a method called reverseByN that takes a queue of integers and an integer n as parameters and that reverses each successive sequence of length n in the queue. For example, suppose that a variable called q stores the following sequence of values:

```
front [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15] back
```

and we make the following call:

```
reverseByN(q, 3);
```

Then q should store the following values after the call:

```
front [3, 2, 1, 6, 5, 4, 9, 8, 7, 12, 11, 10, 15, 14, 13] back
```

Notice that the first three values (1, 2, 3) have been reversed, as have the next three values (4, 5, 6), the next three values (7, 8, 9), and so on. If the size of the queue is not an even multiple of n, then there will be a sequence of fewer than n values at the end. This sequence should be reversed as well. For example, if q stores this sequence:

```
front [8, 9, 15, 27, -3, 14, 42, 8, 73, 19] back
```

and we make the call:

```
reverseByN(q, 4);
```

Then q should store the following values after the call:

```
front [27, 15, 9, 8, 8, 42, 14, -3, 19, 73] back
```

Notice that the two sequences of length 4 have been reversed along with the sequence of two values at the end (73, 19). If n is greater than the size of the queue, then the method should reverse the entire sequence.

You are to use one stack as auxiliary storage to solve this problem.