Questions

1. Recall the insertion-sort method from lecture 3 which sorts an array from smallest to largest. Show the contents of the array after each pass through the `for` loop.

```
index   initial    after 1    after 2
-----   -------    -------    -------    .....
  0       3.2
  1       4.1
  2      -1.0
  3       6.0
  4      -5.6
```

2. Suppose you are given a doubly linked link class `DLinkedList` written in Java. Suppose the class has a method '`DNode remove(DNode node)`' which removes a given `node`. Assume the `DLinkedList` had fields `header` and `tailer` which are dummy nodes, as discussed in the lectures. (See also the online code.)

   Fill in the missing code below for a second (i.e. overloaded) `remove` method that takes as input a positive integer `n`, removes the `n`-th node from the list (where n can be a number from 1 to `size`), and returns a `boolean` which indicates whether the remove was successful or not. In particular, `remove(n)` returns `true` if the list has at least `n` elements and it returns `false` otherwise.

```
public boolean remove(int n){    //  NOTE: valid indices are from 1 to n
    if (n <= 0)  return false;   //  (not 0 to n-1)  which is different
    DNode cur = header;          //  from what you are used to.
    int i = 0;
    while (cur.next != tailer){

                //   add code here

        this.remove(cur);

                //   add code here
        }
    }
    return false;
}
```

3. Consider the Java code:

```
public void display( ArrayList<E>  list ){
  E  e;
  for (int i = 0;  i < list.size(); i++){
    e = list.get(i);
    System.out.println( e.toString() );
  }
}
```

   (a) How does the number of steps of this method depend on n, the number of elements in the list, i.e. the value returned by method `size()` ? We are interested in the dependence on n here and not on various constants factors c_i.

   (b) Would your answer change if `list` were declared to be type `LinkedList<E>` ? If so, how? If not, why not?

4. Consider the following sequence of stack operations:

   `push(d), push(h), pop(), push(f), push(s), pop(), pop(), push(m).`

   (a) Assume the stack is initially empty, what is the sequence of popped values, and what is the final state of the stack? (Identify which end is the top of the stack.)

   (b) Suppose you were to replace the `push` and `pop` operations with `enqueue` and `dequeue` respectively. What would be the sequence of dequeued values, and what would be the final state of the queue? (Identify which end is the front of the queue.)

5. Use a stack to test for balanced parentheses, when scanning the following expressions. Your solution should show the state of the stack each time it is modified. (The "state of the stack" must indicate which is the top element i.e. least recently added.)

   Only consider the parentheses `[,],(,),{,}` . Ignore the variables and operators.

   (a) `[ a + { b / ( c - d ) + e / (f + g ) } - h ]`

   (b) `[ a { b + [ c ( d + e ) - f ] + g }`

6. Suppose you have a stack in which the values 1 through 5 must be pushed on the stack in that order, but that an item on the stack can be popped at any time. Give a sequence of push and pop operations such that the values are popped in the following order:

   (a) 2, 4, 5, 3, 1

   (b) 1, 5, 4, 2, 3

(c) 1, 3, 5, 4, 2

(It might not be possible in each case.)

7. (a) Suppose you have three stacks s1, s2, s2 with starting configuration shown on the left, and finishing condition shown on the right. Give a sequence of push and pop operations that take you from start to finish.

(b) What if the finish configuration on s3 was BDAC (with B on top) ?

8. Assume you have a stack with operations: `push()`, `pop()`, `isEmpty()`. How would you use these stack operations to simulate a queue? In particular, how would you simulate operations `enqueue()` and `dequeue()`?

   Hint: use two stacks.

9. Assume you have a queue with operations: `enqueue()`, `dequeue()`, `isEmpty()`. How would you use the queue methods to simulate a stack ? In particular, how would you simulate `push()` and pop() ?

   Hint: use two queues.

Solutions

1. The * indicates that the array is sorted up to that element.

```
index   initial    after 1    after 2    after 3    after 4
-----   -------    -------    -------    -------    -------
  0       3.2        3.2       -1.0       -1.0       -5.6
  1       4.1        4.1*       3.2        3.2       -1.0
  2      -1.0       -1.0        4.1*       4.1        3.2
  3       6.0        6.0        6.0        6.0*       4.1
  4      -5.6       -5.6       -5.6       -5.6        6.0*
```

2.
```
public boolean remove(int n){
    if (n <= 0)  return false;
    DNode cur = header;
    int i = 0;
    while (cur.next != tailer){

      i++;                    //   SOLUTION: ADD THIS LINE
      cur = cur.next;         //   SOLUTION: ADD THIS LINE
      if (i == n){            //   SOLUTION: ADD THIS LINE

        this.remove(cur);

        return true;          //   SOLUTION: ADD THIS LINE
        }                     //   SOLUTION: ADD THIS LINE
    }
    return false;
}
```

3. Let $t(n)$ be the number of steps executed.

   (a) `get(i)` can be done in constant time (independent of n) for an array (and `e.toString()` is constant time too, of course). We loop through a list of n elements. Thus, for some constant c,
   $$t(n) \approx cn.$$

   (b) For a linked list, `get(i)` requires time proportional to i, since you have to step from the front of the list to the ith element. Thus, the time required to *get* each of the elements will be roughly a sum
   $$t(n) \approx c(1 + 2 + 3 + \cdots + n) = c\frac{n(n+1)}{2}.$$
   for some constant c.

4. (a) Sequence of popped values: h,s,f. State of stack (from top to bottom): m, d
   (b) Sequence of dequeued values: d,h,f. State of queue (from front to back): s,m.

5. (a)
```
-        means empty stack
[
[{
[{(      TOP OF STACK IS ON THE RIGHT
[{
[{(
[{
[
-        empty stack, so brackets match
```
   (b)
```
-
[
[{
[{[         TOP OF STACK IS ON THE RIGHT
[{[(
[{[
[{
[           stack not empty, so brackets don't match
```

6.

```
24531          15423               13542

push 1         push 1              push 1
push 2         pop                 pop
pop            push 2              push 2
push 3         push 3              push 3
push 4         push 4              pop
pop            push 5              push 4
push 5         pop                 push 5
pop            pop                 pop
pop             x                  pop
pop           (not possible)       pop
```

7.

(a)

```
s2.push( s1.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s2.pop() )
s3.push( s2.pop() )
```

(b)

```
s2.push( s1.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s1.push( s2.pop() )
s3.push( s2.pop() )
s2.push( s1.pop() )
s3.push( s1.pop() )
s3.push( s2.pop() )
```

8.

The isEmpty() method is the same for a stack or queue. There is nothing to be done here.

For the enqueue and dequeue operations, I present two solutions.

Solution 1

The first solution is to implement `enqueue` by just pushing the new element on top of the stack. In this solution, the bottom of the stack is the front of the queue and the top of the stack is the back of the queue. How can we implement `dequeue`, that is, how do we remove the bottom element of the stack?

The idea is to use two stacks s and $tmpS$. We first pop all items from the original stack s, pushing each popped element directly onto the second stack $tmpS$. We then pop the top element of the second stack (which is the oldest element, hence it is the element to be dequeued). Finally, we refill the stack s by popping all elements from the second stack, pushing each back on to the first stack.

```
ALGORITHM: enqueue (using a stack operations to simulate the queue)
INPUT:  a stack s and new element e\\
OUTPUT:  the stack s with the new element e at the top \\

  s.push(e)
```

```
ALGORITHM: dequeue (using a stack operations to simulate the queue)
INPUT:   a stack s
OUTPUT:  the stack s with the bottom  element removed

  tmpS <-  new empty stack
  while !s.isEmpty()
    tmpS.push( s.pop() )
  returnValue <- tmpS.pop()                  //  the dequeued element
  while  !(tmpS.isEmpty())
    s.push( tmpS.pop() )
  return  returnValue
```

Solution 2

Here we let `dequeue` be simple and just pop the stack. For this to work, the stack needs to store the elements such that the oldest element is on top of the stack.

```
ALGORITHM: dequeue (using a stack operations to simulate the queue)
INPUT:  a stack s with the oldest element on top
OUTPUT:  the oldest element (and the stack s should no longer contain
         that element)

  return s.pop()
```

With this solution, `enqueue(e)` needs to do the heavy lifting: `enqueue` uses a temporary stack to invert the order of elements currently in the stack, so that the newest element is on top of this temporary stack. Then it pushes the new element on top of the temporary stack. Then it inverts the stack again i.e. recreates the original stack, but now the the newest element which was added is on the bottom.

```
ALGORITHM: enqueue (using a stack operations to simulate the queue)
INPUT:  a stack s with the oldest queue element on top, and a new
        element e to be enqueued
OUTPUT: the stack s with e inserted at the bottom

 tmp <- new empty stack
 while  ! (s.isEmpty())
   tmpS.push( s.pop() )
 tmpS.push(e)
 while ! (tmpS.isEmpty())
   s.push( tmpS.pop())
```

9. The concepts are similar to the previous question.

Solution 1

```
ALGORITHM:  push  (using queue operations enqueue, dequeue, isEmpty)
INPUT:  a queue and a new element e
OUTPUT:  the queue q with the newest element e added

  q.enqueue(e)



ALGORITHM:  pop  (using queue operations enqueue, dequeue, isEmpty)
INPUT:  a queue q
OUTPUT:  the newest element (and the queue q should no longer contain
         this element)

 tmpQ <-  new empty queue
 while   ! (q.isEmpty())
   tmpE  <-   q.dequeue()
   if   ! (q.isEmpty())
      tmpQ.enqueue( tmpE )
   else
      while   !( tmpQ.isEmpty())
         q.enqueue( tmpQ.dequeue())
   return tmpE
```

Solution 2

Here the idea is similar, but now push does the heavy lifting.

```
ALGORITHM: pop (using queue operations enqueue, dequeue, isEmpty )
INPUT:   a queue q
OUTPUT:  the newest element in q (and the queue q no longer contains
         that element)

 q.dequeue()



ALGORITHM: push (using queue operations enqueue, dequeue, isEmpty )
INPUT:   a queue q and new element e to be added
OUTPUT:  the queue q with the new element added at the front of the
         queue (so that pop will return it)
```

```
make a new empty queue qTmp
qTmp.enqueue(e)
while !q.isEmpty()
   qTmp.enqueue( q.dequeue() )
q <- qTmp
return qTmp
```