Data Structures 2018
Exercise 3, solutions (Week 39)

1.-2.

```
/*
   void insertLast (Object o)
   inserts object o to the back of the queue
*/
public void insertLast (Object o)
{
  DoubleLinkNode n = new DoubleLinkNode (o);
  if (IsEmpty ())
    head = tail = n;
  else
    {
      n.prev = tail;
      tail.next = n;
      tail = n;
    }
}

/*
   Object removeLast ()
   Removes and returns the last object in the queue, otherwise returns null
*/
public Object removeLast ()
{
  Object result = null;
  if (! IsEmpty ())
    {
      result = tail.data;
      tail = tail.prev;
      if (tail != null)
        tail.next = null;
      else
        head = null;
    }
  return result;
}
```

3.

```
/*
   void Reverse ()
   reverses the order of the queue's elements
*/
public void reverse ()
{
  DoubleLinkNode tmp;
  for (DoubleLinkNode i = head; i != null; i = i.prev)
    {
      tmp = i.next;
```

```
      i.next = i.prev;
      i.prev = tmp;
    }
  tmp = head;
  head = tail;
  tail = tmp;
}
```

The complexity of the algorithm is $\mathcal{O}(n)$.

4. a) $\mathcal{O}(n)$
   b) $\mathcal{O}(n^2)$
   c) $\mathcal{O}(n)$
   d) $\mathcal{O}(n^2+m)$ (It should be noted that n and m are independent. Thus one can not simply deduce that the complexity is $\mathcal{O}(n^2)$. This is because it might be that $m = n^3$, in which case the complexity would be $\mathcal{O}(n^3)$.)

5. In these solutions the top element of the stack is on the right and the first element of the queue is on the left.
   a) $S = (a, b)$
   b) $S = (a, a, b, b)$
   c) $Q = (b, c)$
   d) $S = (a, a)$, $Q = (a)$

6. A queue can be implemented with two stacks. Let us consider stacks A and B. A new element is pushed to stack A and a variable called *count* keeps the track of the number of elements in the stack. When the first element of the queue is to be removed (from the bottom of stack A), we can use the following steps:
   - Transfer $count - 1$ elements from stack A to stack B.
   - Return and pop the last element in stack A.
   - Transfer elements from stack B to stack A.

   The pseudocodes can be found below (Enqueue and Dequeue):
   Insertion of a new element can be done in constant time ($\mathcal{O}(1)$). However, the complexity of the dequeue operation is $\mathcal{O}(n)$.

7. Algorithm 1.
   While computing the moving average, a circular buffer is usually used instead of an ordinary queue. A circular buffer is a data structure in which the addition of a new element automatically removes the oldest element. In other words, when using a circular buffer, the enqueue-operation adds a new element, removes the oldest element, and returns the removed element. There-

Enqueue(x)
  A.Push(x)
  $count \leftarrow count + 1$
Dequeue(x)
  $count \leftarrow count - 1$
  for $i \leftarrow 1$ **to** $count$ **do**
    B.Push(A.Pop())
  end for
  $value \leftarrow A.pop()$
  for $i \leftarrow 1$ **to** $count$ **do**
    A.Push(B.Pop())
  end for
  return $value$

Algorithm 1 Computes the moving average. Q is a queue containing n previous input values and S is their average. In the beginning, queue Q is initialized in such a way that all n elements are equal to zero.

ComputeMovingAverage(X,n)
  $Q \leftarrow$ new Queue(n)
  $R \leftarrow$ new Queue()
  $S \leftarrow 0$
  $x \leftarrow X$.head()
  while $x \neq$ null **do**
    $S \leftarrow$ MovingAverage($x.val$, Q.length(), Q, S)
    R.enqueue(S)
    $x \leftarrow x$.next()
  end while
  return R
MovingAverage (x, n, Q, S)
  $x \leftarrow x/n$
  $S \leftarrow S + x - Q$.dequeue()
  Q.enqueue(x)
  return S

fore, if circular buffering was applied here, the dequeue-operation on the second line of MovingAverage could be replaced with the enqueue-operation from the third line and the third line could be removed.

8. Algorithm 2.

Algorithm 2 Sum of integer lists.

```
Sum(A, B)
  a ← A.head
  b ← B.head
  R ← new list
  R.head ← new node
  r ← R.head
  c ← a.val + b.val
  r.val ← c mod 10
  c ← ⌊c/10⌋
  a ← a.next
  b ← b.next
  while a ≠ null and b ≠ null do
    r.next ← new node
    r ← r.next
    c ← c + a.val + b.val
    r.val ← c mod 10
    c ← ⌊c/10⌋
    a ← a.next
    b ← b.next
  end while
  if b ≠ null then
    a ← b
  end if
  while a ≠ null do
    r.next ← new node
    r ← r.next
    c ← c + a.val
    r.val ← c mod 10
    a ← a.next
    c ← ⌊c/10⌋
  end while
  if c > 0 then
    r.next ← new node
    r.next.val ← c mod 10
  end if
  return R
```