CPSC 331 — Solutions for Tutorial Exercise #8
Basic Abstract Data Types: Implementations of Stacks and Queues

1. After reviewing the description of a ***linked-list based implementation of a queue***, you were asked to write pseudocode for each of the following operations. You were asked to explain why the algorithm you described is correct and to confirm (assuming the uniform cost criterion) that the execution of each operation requires at most a constant number of steps in the worst case.

   Note: Recall that a singly linked list can be used to represent a queue by storing the elements of the queue, on the list, in order from the earliest existing element to the most recently added (existing) element — so that the element at the ***front*** of the queue is at the ***head*** of the linked list, and the element at the ***rear*** of the queue is at the ***tail*** of the linked list.

   (a) Initialization of an empty queue.

   Solution: It is necessary and sufficient to create an empty list by setting its `front` and `rear` (which could also have been called its `head` and `tail`, respectively) appropriately:

   1. `front = null`
   2. `rear = null`

   This is correct because it creates an empty singly linked list, which represents an empty queue. It certainly uses at most a constant number of steps — namely, two of them — in the worst case, assuming the uniform cost criterion.

   (b) Insertion of an element `x` onto the queue.

   Solution: It is necessary and sufficient to insert the input element `x` onto the ***rear*** of the singly linked list in order to ensure that the elements of the queue are organized in order from the earliest remaining element to the most recently inserted (remaining) element, as required.

   The case the queue (and linked list) was initially empty should be treated as a special case: In this case both the ***front*** and ***rear*** of the list should be set to be a new node storing `x` (whose ***next*** node is `null`). Otherwise, a new node storing `x`, whose ***next*** node is `null`, should be set to be the ***next*** node for the current ***rear*** of the list, and the ***rear*** of the list should then be set to be this new node:

```
insert(x) {
```
1. Create a new node whose value is `x` and whose ***next*** node is `null`.
2. `if (front == null) {` // List is empty if and only if this test passes
3. Set `front` to be this new node.
4. Set `rear` to be this new node.

```
} else {
```
5. Set the ***next*** node of the current `rear` to be the new node.
6. Set `rear` to be the new node.

```
}
```

A consideration of the special and general cases should suffice to establish that this algorithm is correct. An examination of code confirms that at most ***four*** steps are carried out, so the execution of this algorithm uses at most constant time in the worst case, as required.

(c) Peeking at the element of the front of the queue.

Solution: Since the element at the front of the queue is stored at the ***front*** of the list it is necessary, and sufficient, to return the value stored at this node — throwing a `NoSuchElementException`, instead, if the list is empty (and `front` is `null`):

```
peek() {
```
1. `if` $(\texttt{front} \neq \texttt{null})$ `{`
2. Return the value stored at `front`.

```
} else {
```
3. Throw a `NoSuchElementException`.

```
}
```

Correctness follows from the organization of the elements of the queue on the linked list. Since at most ***two*** steps are carried out, an execution of this algorithm also uses at most a constant number of steps in the worst case.

(d) Removal of the element at the front of the queue.

Solution: In the most general case the queue is not empty and at least two elements are stored on it. A consideration of the organization of the elements of the queue on the linked list confirms that a removal is correctly carried out, in this case, if the value stored at the current `front` of the linked list is returned after resetting `front` to the ***next*** node of the current `front`.

In one special case, the queue stores a single element before this operation is applied. In this case the value stored at the `front` of the queue should be returned and — since both the queue and linked list should now be empty — both `front` and `rear` should be set to be `null`, in order for ensure that the algorithm also correctly solves the desired problem in this case.

In the only other case, the queue (and linked list representing it) is empty before this operation is applied, so a `NoSuchElementException` should be thrown in order to ensure that the problem is correctly solved.

```
remove() {
1.  if (front ≠ null) {
2.    Set frontValue to be the value stored at front.
3.    if (the next node after front is not null) {
4.      Set front to be the next node after the current front.
      } else {
5.      Set front to be null.
6.      Set rear to be null.
      }
    } else {
7.    Throw a NoSuchElementException.
    }
}
```

An examination of the pseudocode confirms that at most ***five*** steps are carried out during an execution of this algorithm, so it also uses at most a constant number of steps in the worst case.

(e) Checking to see whether the queue is empty.

Solution: Since the queue is empty if and only if the linked list representing it is empty, it suffices to check whether `front` is `null` — returning `true` if this is the case, and returning `false` otherwise:

```
isEmpty() {
1.  if (front == null) {
2.    return true
    } else {
3.    return false
    }
}
```

An examination of the pseudocode confirms that at most ***two*** steps are carried out during an execution of this algorithm, so that it also uses at most a constant number of steps in the worst case.

2. Source code for a `ListQueue.java` program is now available as a supplement for this exercise on D2L.

3. Source code for an `ArrayQueue.java` program is also available as a supplement for this exercise on D2L.

4. In this question you were asked to describe the most significant change needed, for a linked list-based implementation of a queue, if the ordering of the elements on the list was reversed.

   Solution: The operation to `insert` an element onto the queue would not significantly change: While the new element would be need to be added to the ***front*** of the list instead of the ***rear*** of it when implementing a ***queue***, this could be carried out. Indeed, all you would need to do is use the implementation of a `push` operation for a stack.

   Similarly, all you would need to do is replace the implementation of `peek` for a list-based queue with the implementation of the same operation for a list-based stack. This could still be implemented using at most a constant number of steps in the worst case.

   The implementation of `isEmpty` would not need to be changed at all.

   Unfortunately, changes to needed to implement `remove` for a ***queue*** would be considerably more complicated: Since the element to be deleted from the list would now be at the ***rear*** of the list instead of at the ***front*** of it, the node that should then become the new ***rear*** of the list (and whose ***next*** node should now be changed to `null`) would be the node immediately *before* the former rear element, instead of a node *after* it. If no other changes to the linked list are being made then the only way to find the new rear of the list would be to begin at the `front` of the list and work toward the rear, keeping track of the last two nodes visited, until one of them is `null`. The number of steps needed would then be linear in the current size of the data structure, instead of bounded by a constant.

5. The deprecated class called "Stack", that is part of the ***Java Collections Framework***, was described. You were asked to describe why the number of steps needed to `push` an element onto a stack is almost certainly ***not*** bounded by a constant, in the worst case, when this class (and its implementation) is used.

   Solution: The "Stack" class in Java extends another deprecated class called `Vector` and is, therefore, almost certainly array-based: The implementation makes use of an older version of an `ArrayList` instead of a linked list. Consequently an `array`, whose size is the current `capacity` of the data structure, is being used to store the elements on the stack.

   As described in Lecture #7, it is occasionally necessary to ***replace*** the underlying array with another larger one — generally one that is twice as large as the one being replaced. If data must be copied from the old array to the new one then the cost for this — infrequent — operation can be as high as linear in the current size of the data structure.