# Handout 3
# Verified Queue

Construction and Verification of Software

Nova School of Science and Technology
Mário Pereira `mjp.pereira@fct.unl.pt`

Version of November 29, 2023

The goal of this handout is to verify, using Separation Logic in VeriFast, a Java implementation of a Queue, *i.e.*, a FIFO data structure. The queue implementation will be dealing with provides in-place modification, featuring constant time execution for both *push* and *pop* operations. A good description of the behavior of a Queue can be found on the Wikipedia page: `https://en.wikipedia.org/wiki/Queue_(abstract_data_type)`. An even better presentation of such FIFO data structures is given by R. Sedgewick and K. Wayne in their masterpiece *Algorithms* [**?**, Chapter 1.3]. For reference, we provide a complete Java implementation of a FIFO Queue in Appendix A.

During this handout you are supposed to complete the specification and proof given in the companion `.java` file. This includes:

- definition of predicates
- pre-conditions (*i.e.*, `requires` clauses)
- post-conditions (*i.e.*, `ensures` clauses)
- use of `open`/`close` primitives
- calling auxiliary lemmas
- loop invariants

The Queue data type and corresponding operations are already implemented. **Do not change such implementations**.

## 1 Data Type Implementation

The given implementation of queues is efficient, both in space and time. Internally, a queue is just a heap-allocated singly-linked list that follows a FIFO discipline. The `Queue` implementation uses the `Cell` class to represent a heap-allocated node of the data structure. Each `Cell`, contains a field `next` that represents the next element in the list-based representation, as well as the field `content` to store an integer value associated with each node. The representation invariant of class `Cell` is already provided in the companion file. An object of class `Queue` contains two fields: `first`, which points to the first element in the queue; `last`, pointing to the last element of the queue. Finally, field `length` denotes the number of elements currently stored in the queue.

## 2 Queue Representation Invariant

**Exercise 1.** Define a predicate `QueueInv` that captures the *representation invariant* for a well-formed queue. Such predicate should have the following signature:

```
//@    predicate QueueInv(Queue q; list<int> elems) = ...
```

where `q` stands for the entry pointer to the data structure and `elems` is the logical sequence of elements store in the queue.

The `QueueInv` predicate should capture the fact that, **when the queue is not empty**, there is a list segment from `first` to `last`. Also, note that next cell of `last` is always the `null` pointer. □

# 3 Verified Queue Operations

## 3.1 Constructor

The constructor for this ADT is fairly simple: it initializes fields `first` and `last` with the `null` pointer. Initially, the queue contains no elements, hence the initialization of field `length` with `0`.

**Exercise 2.** Complete the specification of the `Queue` constructor. □

## 3.2 Clear the Contents of a Queue

**Exercise 3.** Complete the specification of method `clear`, which clears the contents of a queue. □

## 3.3 Add a New Element

**Exercise 4.** Complete the specification of method `add`, which inserts a new element in the queue. The implementation of this method might seem odd at first sight, with some unnecessary redundancy. This is only to make the proof easier.

**Note:** this is likely to be the most challenging method in all the handout, featuring many `open/close` clauses and calls to auxiliary lemmas. □

## 3.4 Emptiness Test

**Exercise 5.** Complete the specification of method `isEmpty`, which checks whether the queue contains no element. □

## 3.5 Removing an Element

**Exercise 6.** Complete the specification of method `take`, which removes an element from the queue. □

## 3.6 Client Code

**Exercise 7.** A small client class is provided, which contains only the static method `transfer`. Such method exercises the `Queue` class API and implements an operation that appends all elements of `q2` (in order) to the end of `q1`.

Complete the specification of method `transfer`. This includes not only the method pre- and post-conditions of the method, but also the loop invariant. □

# A Java Implementation of FIFO Queues

```
class Cell
{
    Cell next;
    int content;

    public Cell (int v, Cell next) {
        this.next = next;
        content = v;
    }
}

public class Queue
{
    Cell first;
    Cell last;
    int length;

    public Queue () {
        first = null;
        last = null;
        length = 0;
    }

    public void add (int x) {
        Cell cell = new Cell(x, null);

        if (last == null) {
            last = cell;
            first = cell;
            length = 1;
        }
        else {
            if (last == first) {
                last.next = cell;
                last = cell;
                length = 2;
            }
            else {
                last.next = cell;
                last = cell;
                length = length + 1;
            }
        }
    }

    public boolean isEmpty () {
        return length == 0;
    }

    public int take () {
        if (first == last) {
            int c = first.content;
            this.clear();
```

```
            return c;
        }
        else {
            int c = first.content;
            length = length - 1;
            first = first.next;

            return c;
        }
    }
}

class Client
{
    public static void transfer (Queue q1, Queue q2)
    {
      while (!q2.isEmpty())
      {
        int v = q2.take();
        q1.add(v);
      }
    }
}
```