# Learning to Program with F#
Exercises
Department of Computer Science
University of Copenhagen

Jon Sporring, Martin Elsman, Torben Mogensen, Christina Lioma

November 21, 2022

## 0.1 Queue

### 0.1.1 Teacher's guide

**Emne** rekursion, grafik og winforms

**Sværhedsgrad** Middel

### 0.1.2 Introduction

In the following, you are to work with the abstract datatype known as a *queue*. A queue is a a sequence of elements that supports the following operations: checking whether the sequence is empty; removing an element from the front ("left"); adding an element at the end ("right"). Queues appear often in real life: The line[1] waiting for service at a shop counter, orders to be filled in a warehouse, students waiting to be examined at an oral examination.

The abstract datatype of *purely functional queues* consists of:

- the types named `element` and `queue`, where `element` is the type of elements and `queue` the type of queues with such elements;
- the following value and two functions

  ```
  // the empty queue
  emptyQueue: queue
  // add an element at the end of a queue
  enqueue: element -> queue -> queue
  // remove and return the element at the front of a queue
  // precondition: input queue is not empty
  dequeue: queue -> element * queue
  // check if a queue is empty
  isEmpty: queue -> bool
  ```

- the properties these operations must satisfy and that another programmer can rely on, such as queuing an element on an empty queue and then dequeuing from it yields the element added at first and leaves an empty queue behind; or, more generally, first repeatedly queuing elements and then dequeuing until the queue is empty yields the same elements.[2]

These queues are called *(purely) functional* because the enqueue and dequeue operations return a *new* queue whenever they are called, without destroying the old queue. For example, adding an element $e_1$ to a queue $q_0$ of length 15 results in a queue $q_1$ of length 16; then adding another element $e_2$ to $q_0$ also results in a queue $q_2$ of length 16, but one that is different from $q_1$ in its last element. At this point we have 3 separate queues, each of which we can used in future operations: $q_0$, $q_1$ and $q_2$.[3]

[1]In American English. Called indeed *queue* in British English.

[2]This can be expressed as a precise mathematical property, which in turn can be used to systematically test one's implementation to find errors. We'll do this only informally here.

[3]There are also *ephemeral* (also called *imperative*) queues, where enqueue and replace the original queue with the new queue such that there is always just one "current" queue that changes over time. Ephemeral queues have more limited functionality and are easier to implement efficiently using imperative data structures, which we will encounter later in the course.

In this exercise, you will work with functional queues in F#. We'll omit writing "functional" below.

### 0.1.3 Exercise(s)

**0.1.3.1:** In the following you are to create a dotnet project for the assignment.

(a) Using the `dotnet` command line tool, create a new `F#` console application named `5i`.
`dotnet new console -lang "F#" -o 5i`

(b) Rename the file `Program.fs` to `testQueues.fs`.

(c) Update `5i.fsproj` to compile `testQueues.fs` instead of `Program.fs`.

(d) Ensure the project works by running `dotnet build` and `dotnet run`.

**0.1.3.2:** In the following, you are to implement your own queue module using lists in F# to represent the (sequence of elements in) a queue. The module is to be called `IntQueue`.

(a) Given the description of the abstract datatype Queue above, write a signature file `intQueue.fsi` for the functional queues.

(b) Write an implementation file `intQueue.fs`, implementing the signature file above using lists in F# where the elements are F# integers.

(c) Add `intQueue.fsi` and `intQueue.fs` to `5i.fsproj`,
so they are compiled *before* `testQueues.fs`.

(d) In `testQueues.fs`, show your implementation works by using your queue. As a minimum, you should add the following series of tests of your `IntQueue` module:
changed from v1: negate the result of `IntQueue.isEmpty q1` so `nonEmptyTestResult` is `true` when the implementation of `IntQueue.isEmpty` is correct.

```
let intQueueTests () =
    let q0 = IntQueue.emptyQueue
    let emptyTestResult = IntQueue.isEmpty q0
    emptyTestResult
    |> printfn "An empty queue is empty: %A"

    let e1,e2,e3 = 1,2,3
    let q1 = q0 |> IntQueue.enqueue e1
                |> IntQueue.enqueue e2
                |> IntQueue.enqueue e3
    let nonEmptyTestResult = not (IntQueue.isEmpty q1)

    nonEmptyTestResult
    |> printfn "A queue with elements is not empty: %A"

    let (e,q2) = IntQueue.dequeue q1
    let dequeueTestResult = e = e1
    dequeueTestResult
    |> printfn "First in is first out: %A"

    let allTestResults =
        emptyTestResult &&
        nonEmptyTestResult &&
        dequeueTestResult

    allTestResults
    |> printfn "All IntQueue tests passed: %A"
    // Return the test results as a boolean
    allTestResults

// Run the IntQueue tests
let intQueueTestResults = intQueueTests ()
```

**0.1.3.3:** A problem with the queue specification above is that there is a precondition on the dequeue operation: A programmer must always ensure that the argument to dequeue is nonempty before calling dequeue. In other words, even though the F# type system does not flag it as an error, it *is* an error (by the programmer) to call dequeue with the empty queue.

In the following, you are to implement another version of your queue module using lists in F# to represent the (sequence of elements in) a queue, with error handling. The module is to be called `SafeIntQueue`, the signature file `safeIntQueue.fsi` and the implementation file `safeIntQueue.fs`.

Change the queue specification such that `SafeIntQueue.dequeue` returns an `(element option) * queue` value and remove the precondition. Add the new module to `5i.fsproj` and in `testQueues.fs`, add a corresponding test suite that shows your implementation works; that is, its operations perform queuing and dequeuing, and, additionally `SafeIntQueue.dequeue(emptyQueue)` returns `None`.

**0.1.3.4:** A queue is an abstract datatype, and as such the operations of a queue should not depend on the type of elements. In the following you are to implement a *generic queue*, so it is possible to create queues of any type, e.g. queues of `int`, queues of `float`, queues of `string` or even queues of `queue`.

The module is to be called `Queue`, the signature file `queue.fsi` and the implementation file `queue.fs`. Add the new module to `5i.fsproj` and in `testQueues.fs`, add a corresponding test suite that shows your implementation works; that is, its operations perform queuing and dequeuing, and additionally that you can build queues of different types with the same generic library. As a minimum, demonstrate queues of `int`, `float` and `string`.