Imperative Programming (Parts 1&2): Sheet 5

Joe Pitt-Francis

Most questions by Mike Spivey, adapted by Gavin Lowe

Suitable for around Week 6, Hilary term, 2019

- Any question marked with † is a self-study question and an answer is provided at the end. Unless your tutor says otherwise, you should mark your attempts at these questions, and hand them in with the rest of your answers.
- Questions marked [**Programming**] are specifically designed to be implemented. Unless your tutor says otherwise, provide your tutor with evidence that you have a working implementation.
- You should give an invariant for each non-trivial loop you write, together with appropriate justification.

Question 1

[**Programming**] A linked list representing an `Int` sequence can be made using the class `Node`:

```
class Node(val datum: Int, var next: Node)
```

where `datum` represents the value of this node and `next` is a reference to the next node in the linked list (or `null` if there are no nodes following).

(a) Write a loop which will produce a list `myList` by creating `new` Nodes containing the numbers 1,2,. . . 12 and adding them to the head of a list. Adding to the head of a list mimics Haskell's cons operator ":".

(b) In the `Node` class, override Scala's `toString` operation so that, instead of printing a given `Node` as a reference, it is able to print the value of the list (the value of the current node and its successors).

```
class Node(val datum: Int, var next: Node){
    override def toString : String = ...
}
```

```
//... then with myList defined as above
scala> println("List is "+myList)
List is 12 -> 11 -> 10 -> 9 -> 8 -> 7 -> 6 -> 5 -> 4 -> 3 -> 2 -> 1.
```

(c) Write a `while` loop which will reverse the list in linear time (without creating any new `Node` objects). Linked list algorithms can be tricky to debug...so do not forget to write down a loop invariant.

Question 2 †

Consider the `store` operation in the implementation of the phone book using a linked list with a dummy header (`LinkedListHeaderBook.scala`). If the name was not already in the book, we added it to the *front* of the list. Change the implementation so that the name is added to the *end* of the list.

Question 3

Show how to implement the phone book (including `delete`) using a singly-linked list with a list header, and where the list is stored in order of the names. Give an appropriate abstraction function and datatype invariant.

Question 4

(a) If a phone book contains names $s_0, s_1, \ldots, s_{n-1}$ and a random `recall` request matches them with probabilities $p_0, p_1, \ldots, p_{n-1}$, matching none of them with probability $q = 1 - (p_0 + \ldots + p_{n-1})$, show that the expected amount of work done in searching the phone book linearly is minimised if the names are arranged in such an order that $p_0 \geq p_1 \geq p_2 \geq \ldots \geq p_{n-1}$.

(b) When the probability p_i of a given name s_i being requested is unknown, then a good heuristic for fast `recall` with linear searching is to order the data by most-recently-used. This means that, when an entry needed, it is moved to the front of the concrete sequence of entries. Pick one of the implementations of the `Book` trait and outline how this heuristic could be programmed.

Question 5

The state of a queue holding data of type `A` is a sequence of values from `A`. In lectures we saw a specification of a queue and this specification is reproduced in Figure 1.

Implement a class

```
class ArrayQueue extends Queue[Int]{
  val MAX = 100 // max number of pieces of data
  ...
}
```

```
/** A queue of data of type A.
  * state: q : seq A
  * init: q = [] */
trait Queue[A]{
  /** Add x to the back of the queue
    * post: q = q0 ++ [x] */
  def enqueue(x: A)

  /** Remove and return the first element.
    * pre: q ≠ []
    * post: q = tail q0 ∧ returns head q0
    * or post: returns x s.t. q0 = [x] ++ q */
  def dequeue: A

  /** Is the queue empty?
    * post: q = q0 ∧ returns q = [] */
  def isEmpty: Boolean
}
```

Figure 1: An abstract datatyp specification for a queue

that can store at most `MAX` values, using an array `data[0..MAX)` internally to store the data. The idea is that the data is either stored in some contiguous segment `data[i..j)`, or in `data[i..MAX) ++ data[0..j)` for some appropriate `i` and `j`; in the latter case the data wraps around the array. This data structure is sometimes called a *circular array*.

State a suitable datatype invariant and abstraction function. If your implementation does not meet the above specification, explain why and adapt the specification appropriately.

Also specify and implement an operation

```
  /** Is the queue full? */
  def isFull : Boolean
```

Your implementation should ensure that each operation runs in time $O(1)$.

Question 6

The state of a queue holding data of type `A` is a sequence of values from `A`. The abstract datatype is given as a Scala `trait` in Figure 1.

Show how to implement a queue of integers using a class that uses a linked list as its internal data structure. Think carefully about the details of the internal state: it is possible to arrange for all the operations to run in time $O(1)$. Give a suitable abstraction function and datatype invariant.

```
class IntQueue extends Queue[Int]{
  ...
}
```

Question 7

A *double-ended queue* is like a queue, except it allows data to be added to and removed from either end. A double-ended queue storing `Int`s will have the following signature.

```
class DoubleEndedQueue{
  /** Is the queue empty? */
  def isEmpty : Boolean = ...

  /** add x to the start of the queue. */
  def addLeft(x:Int) = ...

  /** get and remove element from the start of the queue. */
  def getLeft : Int = ...

  /** add element to the end of the queue. */
  def addRight(x: Int) = ...

  /** get and remove element from the end of the queue. */
  def getRight : Int = ...
```

Specify the class in terms of an abstract state

state: $s : \mathrm{seq}\, Int$

Give pre- and post-conditions for the operations (use Haskell notation, as appropriate).

We can implement a double-ended queue using a doubly linked list, that is a linked list where each node contains a reference to both the previous and next nodes in the list:

```
  class Node(var datum: Int, var prev: Node, var next: Node)
```

Complete the implementation. Formalise the abstract sequence represented by the linked list using an abstraction function, and state suitable datatype invariants.

Answer to question 2 †

This is fairly easy, because `find` returns the last node of the list in the case of an unsuccessful search.

```
  /** Add the maplet name -> number to the mapping */
  def store(name: String, number: String) = {
    val n = find(name)
    if(n.next == null) // store new info after n
      n.next = new LinkedListHeaderBook.Node(name, number, null)
    else n.next.number = number // as before
  }
```