Tuan Nguen

**Question 1:**

a)

```
object Test{
  def main(args : Array[String]) = {
    var myList : Node = null
    var i = 1
    while(i <= 12){
      val n1 = new Node(i, myList)
      myList = n1
      i += 1
    }
    println("List is " + myList)
}
```

b)

```
class Node(val datum: Int, var next: Node){
  override def toString : String = {
    if(next == null) datum.toString else datum + " -> " + next
  }
}
```

c)

```
// Invariant: All nodes after prev that are reachable with next are
// reversed
//         && nothing is referencing next
//         && current is referencing the first node that is not reversed
var next : Node = null; var current : Node = myList
var prev : Node = null
while(current != null){
  next = current.next
  current.next = prev
  prev = current
  current = next
}
```

**Question 2:**

```
/** Add the name to the phonebook */
def store(name: String, number: String) = {
  val n = find(name)
  if(n.next == null){
    n.next = new LinkedListHeaderBook.Node(name, number, null)
  }
  else n.next.number = number
```

```
}
```

**Question 3:**

```
trait Book{
  // State: book : String -|-> String
  // Init:  book = {}

  // Add the maplet name -> number to the mapping
  // Post: book = book_0 (+) {name -> number}
  def store(name: String, number: String)

  // Return the number stored against name
  // Pre: name in dom book
  // Post: book = book_0 && returns book(name)
  def recall(name: String) : String

  // Is name in the book?
  // Post: book = book_0 && returns name in dom book
  def isInBook(name: String): Boolean

  // Delete the number stored against a name (if it exists)
  // Post: if name is in book: book = book0 - {name} && return true
  //        if not in book   : book = book0 && return false
  def delete(name: String) : Boolean
}

class LinkedListHeaderBook extends Book{
  private var list = new LinkedListHeaderBook.Node("?", "?", null)
  // list represents the mapping composed of (n.name -> n.number)
  // maplets, when n is a node reached by following 1 or more
  // next references.

  /** Return the node before the one containing name.
    * Post: book = book_0 && returns n s.t. n in L(list) &&
    * (n.next.name=name or n.next=null if no such Node exists)*/
  private def find(name:String) : LinkedListHeaderBook.Node = {
    var n = list
    // Invariant: name does not appear in the nodes up to and
    // including n; i.e.,
    // for all n1 in L(list.next, n.next), n1.name != name
    while(n.next != null && n.next.name != name) n = n.next
    n
  }

  /** Is name in the book? */
  def isInBook(name: String): Boolean = find(name).next != null
```

```
  /** Return the number stored against name */
  def recall(name: String) : String = {
    val n = find(name); assert(n.next != null); n.next.number
  }

  /** Add the maplet name -> number to the mapping */
  def store(name: String, number: String) = {
    val n = find(name)
    if(n.next == null){
       // Invariant: obj.name < name && obj.next.name < name
       //         && list = list0 && it is ordered
       var obj = list
       if(obj.next != null && name < obj.next.name){   // If the node
should be added to the head
         list.name = name; list.number = number
         list = new LinkedListHeaderBook.Node("?", "?", list)
       }
       else{
         while(obj.next != null && name > obj.next.name){
           obj = obj.next
         }
         val n1 = new LinkedListHeaderBook.Node(name, number, obj.next)
         obj.next = n1
       }
    }
    else n.next.number = number
  }

  /** Delete the number stored against name (if it exists);
    * return true if the name existed. */
  def delete(name: String) : Boolean = {
    val n = find(name)
    if(n.next != null){ n.next = n.next.next; true }
    else false
  }
}

// Companion object
object LinkedListHeaderBook{
  private class Node(var name: String, var number: String, var next:
Node)
}
```

**Question 4:**

a) The expected amount of work done is

$$\sum_{0}^{n} ip_i$$

Where $np_n$ is going to represent the work when the entry is not in the phonebook. $p_n$ does not depend on how the names are ordered, so we can just ignore it. We have to try to minimize the sum. I will prove that for two entries $a*p_i$ and $b*p_j$, $a < b$, $p_i > p_j$ (a and b are independent from i and j), is the optimal solution.

$$a * p_i + b * p_j < a * p_j + b * p_i \Leftrightarrow$$
$$(a - b) * p_i < (a - b) * p_j \Leftrightarrow$$
$$0 < (a - b)(p_j - p_i)$$

which is true because both of them are negative so the RHS is positive.
This applies to all pair of names, so the sum would be minimized if $p_0 \geq p_1 \geq \cdots \geq p_{n-1}$.

b)

```
/** Return the number stored against name */
def recall(name: String) : String = {
  val i = find(name)
  assert(i < count)
  var swap = entries(i)
  entries(i) = entries(0)
  entries(0) = swap
  entries(0)._2
}
```

The rest of the class is the same as shown in the lectures.

**Question 5:**

```
/** A queue of date of type A.
  * state: q: seqA
  * 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 && return head q0
    * or post: returns x s.t. q0 = [x] ++ q */
  def dequeue : A

  /** Is the queue empty?
    * post: q = q0 && return q == [] */
  def isEmpty : Boolean
```

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

class ArrayQueue extends Queue[Int]{
  val MAX = 5 // max number of pieces of data
  var data = new Array[Int](MAX)
  var i = 0; var j = 0; var count = 0
  /** Abs: queue = data[i..j)                  if i < j
    *           or data[i..MAX) ++ data[0..j) if j < i
    * DTI: i + j <= MAX  && 0 <= count <= MAX
        && 0 <= i < MAX && 0 <= j < MAX */

  /** Add x to the back of the queue */
  def enqueue(x: Int) = {
    data(j) = x
    if(j == MAX) j = 1 else j += 1
    if(count != MAX) count += 1
  }

  /** Remove and return the first element */
  def dequeue : Int = {
    assert(!isEmpty)
    val oldIndex = i
    if(i == MAX - 1) i = 0 else i += 1
    count -= 1
    data(oldIndex)
  }

  /** Is the queue empty */
  def isEmpty : Boolean = {
    count == 0
  }

  /** Is the queue full */
  def isFull : Boolean = {
    count == MAX
  }
}

object Test{
  def main(args: Array[String]) = {
    var queue = new ArrayQueue
    queue.enqueue(5)
    queue.enqueue(6)
    queue.enqueue(6)
```

```
    queue.enqueue(6)
    queue.enqueue(6)
    println(queue.isEmpty)
    println(queue.isFull)
    queue.dequeue
    queue.dequeue
    println(queue.isEmpty)
    queue.dequeue
    queue.dequeue
    queue.dequeue
    println(queue.isEmpty)
  }
}
```

**Question 6:**

```
/** A queue of data of type A
  * state: q: seqA
  * 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 && reutnr head q0
    * or post: return x s.t. q0 = [x] ++ q */
  def dequeue: A

  /** Is the queue emtpy
    * post: q = q0 && return q = [] */
  def isEmpty: Boolean
}

class IntQueue extends Queue[Int]{
  private var list : IntQueue.Node = null
  private var start : IntQueue.Node = list
  private var end : IntQueue.Node = list
  private var count = 0
  /** Abs: queue = L(start), where L is the list of
    * all nodes that can be reached by the next
    * reference from a certain node
    * DTI: start points to the leftmost element
    *   && end points to the rightmost element
    *   && end.next == null
    *   && queue is finite */
```

```
  /** Add x to the back of the queue */
  def enqueue(x: Int) = {
    if(count == 0){  // If the list is empty
      val n1 = new IntQueue.Node(x, null)
      start = n1
      end = n1
    }
    else{
      val n1 = new IntQueue.Node(x, null)
      end.next = n1
      end = n1
    }
    count += 1
  }

  /** Remove and return the first element */
  def dequeue: Int = {
    assert(!isEmpty)
    var oldStart : IntQueue.Node = start
    start = start.next
    count -= 1
    oldStart.datum
  }

  /** Is the queue empty */
  def isEmpty() : Boolean = {
    count == 0
  }
}

// Companion object
object IntQueue{
  private class Node(val datum: Int, var next: Node)
}

object Test{
  def main(args: Array[String]) = {
    var queue = new IntQueue
    println(queue.isEmpty) // true
    queue.enqueue(1)
    queue.enqueue(2)
    queue.enqueue(3)
    queue.enqueue(4)
    println(queue.isEmpty) // false
    queue.dequeue
```

```
    queue.dequeue
    queue.dequeue
    queue.dequeue
    println(queue.isEmpty) // true
  }
}
```

**Question 7:**

```
trait DoubleQueue{
  /** state: s: seq Int
    * init: s = [] */

  /** Is the queue empty
    * post: queue = queue0 && return is the queue empty*/
  def isEmpty : Boolean

  /** Add x to the start of the queue
    * post: queue = {elem} + queue0 */
  def addLeft(x: Int)

  /** Get and remove element from the start of the queue
    * pre: queue != []
    * post: queue = tail queue0 && return head queue0 */
  def getLeft : Int

  /** Add element to the end of the queue
    * post: queue = queue0 + {elem} */
  def addRight(x: Int)

  /** Get and remove element from the end of the queue
    * pre: queue != []
    * post: queue = init queue0 && return last queue0 */
  def getRight : Int
}

class DoubleEndedQueue{
  private var start : DoubleEndedQueue.Node = null
  private var end : DoubleEndedQueue.Node = null
  private var count = 0
  /** Abs: queue = L(start), where L is the list of
    * all nodes that can be reached by the next
    * reference from a certain node
    * DTI: start points to the leftmost element
    *   && end points to the rightmost element
    *   && start.prev = null && end.next = null
    *   && queue is finite */
```

```
  /** Is the queue empty */
  def isEmpty : Boolean = count == 0

  /** Add x to the start of the queue */
  def addLeft(x: Int) = {
    val n1 = new DoubleEndedQueue.Node(x, null, start)
    if(count == 0) end = n1 else start.next = n1
    start = n1
    count += 1
  }

  /** Get and remove element from the start of the queue */
  def getLeft : Int = {
    assert(!isEmpty)
    val oldStart : DoubleEndedQueue.Node = start
    start = start.next
    count -= 1
    oldStart.datum
  }

  /** Add element to the end of the queue */
  def addRight(x: Int) = {
    val n1 = new DoubleEndedQueue.Node(x, end, null)
    if(count == 0) start = n1 else end.next = n1
    end = n1
    count += 1
  }

  /** Get and remove element from the end of the queue */
  def getRight : Int = {
    assert(!isEmpty)
    val oldEnd : DoubleEndedQueue.Node = end
    end = end.prev
    count -= 1
    oldEnd.datum
  }
}

// Companion object
object DoubleEndedQueue{
  private class Node(var datum: Int, var prev: Node, var next: Node)
}

object Test{
  def main(args: Array[String]) = {
```

```
    var queue = new DoubleEndedQueue
    queue.addRight(1)
    queue.addRight(2)
    queue.addRight(3)
    queue.addRight(4)
    println(queue.isEmpty)
    println(queue.getLeft + " ")
    println(queue.getLeft + " ")
    println(queue.getLeft + " ")
    println(queue.getLeft + " ")
  }
}
```