Problem Sheet 5

Question 1

a) Here is the overridden `toString` method

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

        /**
          * Only works if init = dummy node
          * Pre: S != {}
          */
        override def toString : String = {
                if (this.next != null) {
                        if (this.next.next != null) print(this.next.datum + "->
")
                        else print(this.next.datum)
                        this.next.toString()
                }
        }
}
```

b) Here is the code for implementing the loop that produces a linked list:

```
class Node(val datum: Int, var next: Node){
        ...
        def add(e: Int) : Unit = {
                this.datum = e
                this = new Node(0, this)
        }
        ...
}

def main(args : Array[String]) : Unit = {
        var node = new Node()
        for (i <- 1 until 13) {
                node.add(i)
        }
}
```

part c)

```
def reverse(list : Node) : Node = {
        var previous: Node = null
        var current : Node = list
```

```
        var next : Node = null

        // loop invariant : previous should always be the previous current node,
        // and the next should the next current node, given they are not null
        while (current != null) {
                next = current.next
                current.next = previous
                previous = current
                current = next
        }
}
```

Question 2

Here is the adjusted version of the store method which adds to the end of the list instead of at the head.

```
def store(name: String, number: String) : Unit = {
        temp = this.node
        while (temp.next != null) {
                temp = temp.next
        }
        temp.next = new LinkedListHeaderBook.Node(name, number, null)
}
```

Question 3

```
/**
  * init: S = Node(null, null, null) (dummy header)
  */
class Node(var name: String, var number: String, var next: Node)

class PhoneBook {
  private val header: Node = new Node(null, null, null) // List header (dummy
node)

  /** Adds or updates a contact in the phone book
    * Post: S = S_0 union {(name, string)}
    */
  def add(name: String, number: String): Unit = {
    var prev: Node = header
    var current: Node = header.next
    while (current != null && current.name < name) {
      prev = current
      current = current.next
    }
    if (current != null && current.name == name) {
      current.number = number // Update the existing contact
```

```
    } else {
      prev.next = new Node(name, number, current) // Insert a new contact
    }
  }

  /** Finds a contact by name
    * Post: S = S_0 and return Maybe number (haskell notation)
    */
  def find(name: String): Option[String] = {
    var current: Node = header.next
    while (current != null && current.name != name) {
      current = current.next
    }
    if (current != null) Some(current.number) else None
  }

  /** Deletes a contact by name
    * Pre: S != {}
    * Post: S = S \{(name, number)} given name exists in list
    */
  def delete(name: String): Boolean = {
    var prev: Node = header
    var current: Node = header.next
    while (current != null && current.name != name) {
      prev = current
      current = current.next
    }
    if (current != null) {
      prev.next = current.next // Delete the contact
      true
    } else {
      false
    }
  }
}
```

Question 4

For this question, I will need to make a concrete implementation of the `Book` trait, which I will call `MRUBook`, where MRU stands for most recently used.

Here is the implementation of such a class where the heuristic is taken advantage of:

```
// Init : S = null (starts as an empty node)
class MRUBook extends Book {
  class Node(var name: String, var number: String, var next: Node)

  private var head: Node = null
```

```
  /**
    * Post: S = S union {(name, number)}
    */
  override def store(name: String, number: String): Unit = {
    if (!isInBook(name)) {
      head = new Node(name, number, head)
    } else {
      val numberToStore = recall(name) // This moves the node to the front
      head = new Node(name, numberToStore, head)
    }
  }

  /**
    * DTI : Given name exists in the book, the node for name should be brought to
the front. S should remain the same before and after this.
    */
  override def recall(name: String): String = {
    if (head != null && head.name == name) return head.number

    var prev: Node = head
    var current: Node = head.next

    while (current != null && current.name != name) {
      prev = current
      current = current.next
    }

    if (current != null) {
      prev.next = current.next // Remove current from its position
      current.next = head // Move current to the front
      head = current
      current.number
    } else {
      throw new NoSuchElementException(s"No entry found for $name")
    }
  }

  /**
    * DTI: S should remain the same before and after isInBook
    * Post: S = S && return isIn
    */
  override def isInBook(name: String): Boolean = {
    var current: Node = head
    while (current != null) {
      if (current.name == name) return true
      current = current.next
    }
    false
```

```
  }
}
```

Question 5

Given our specifcation:

```
/** 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) : Unit
/** 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
}
```

And that our concrete implementation needs to be capped at the `MAX` size of 100, here is the implemented `ArrayQueue` class:

```
/**
  * Init: Q = [0,0,...,0] where Q.size = 100
  */
class ArrayQueue extends Queue[Int] {
  val MAX = 100 // max number of pieces of data
  private val data = new Array[Int](MAX)
  private var head = 0 // points to the front of the queue
  private var tail = 0 // points to the end of the queue
  private var size = 0 // tracks the number of elements in the queue

  /** Add x to the back of the queue */
  def enqueue(x: Int): Unit = {
    if (!isFull) {
      data(tail) = x
      tail = (tail + 1) % MAX
      size += 1
    } else {
      throw new IllegalStateException("Queue is full")
    }
  }
```

```
  /** Remove and return the first element. */
  def dequeue(): Int = {
    if (!isEmpty) {
      val elem = data(head)
      head = (head + 1) % MAX
      size -= 1
      elem
    } else {
      throw new NoSuchElementException("Queue is empty")
    }
  }

  /** Is the queue empty? */
  def isEmpty: Boolean = size == 0

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

Our Datatype invariant for this class is the following:

- $0 \leq head < MAX$
- $0 \leq tail < MAX$
- $0 \leq size \leq MAX$
- If the queue is not full, `tail` points to the position where the next element will be inserted.
- If the queue is not empty, `head` points to the first element in the queue.
- The queue can wrap around the array, meaning elements can be in `data[head..MAX) ++ data[0..tail)` if `head > tail`

Our abstraction function for our class would look like the following:

$$q = [data[head],\ data[head+1],\ \ldots,\ data[tail-1]] \ \mid\ \text{given that head} < \text{tail, otherwise}$$
$$q = [data[head],\ \ldots,\ data[MAX-1],\ data[0],\ \ldots,\ data[tail-1]]$$

Question 6

```
// Init: Q = []
class IntQueue extends Queue[Int] {
  private var head: Node = null  // Points to the first element in the queue
  private var tail: Node = null  // Points to the last element in the queue

  /** Add x to the back of the queue
    * DTI: (tail - head) == queue.size (essentially the difference between the
tail and the head will give the size, not literally operating a minus on a Node
class)
    * Post: Q = head:xs:tail where head = x
    */
```

```
  def enqueue(x: Int): Unit = {
    val newNode = new Node(x, null)
    if (isEmpty) {
      head = newNode
    } else {
      tail.next = newNode
    }
    tail = newNode
  }

  /** Remove and return the first element
    * DTI: head -> ... -> tail otherwise head == tail (empty list)
    * Post: head -> head_0 -> ...  => return head.data and head = head_0
    */
  def dequeue(): Int = {
    if (isEmpty) throw new NoSuchElementException("Queue is empty")
    val data = head.data
    head = head.next
    if (head == null) tail = null  // If the queue becomes empty
    data
  }

  /** Is the queue empty?
    * Post: return boolean statement (queue is empty)
    */
  def isEmpty: Boolean = head == null
}
```

As for the Datatype Invariant, here are the specifications for that:

- If `head` is `null`, then the queue is empty, and `tail` must also be `null`.
- If `head` is not `null`, then it points to the first node of a singly linked list that terminates in `null`.
- If the queue has only one element, then `head` and `tail` point to the same node.
- For any node `n` in the list, `n.next` is `null` if and only if `n` is the last node in the list, in which case `n` equals `tail`

As for the abstract function: $AF(c) = q$, where q is a sequence $[q_0, q_1, \ldots, q_{n-1}]$ such that q_0 is the integer stored at `c.head`, q_{n-1} is the integer stored at `c.tail`, and q_i is the integer stored at the i^{th} node in the linked list starting from `c.head`. If `c.head` is `null`, then q is the empty sequence $[]$.

Question 7

```
/** A double-ended queue of integers.
  *
  * Abstract State: s: seq Int
```

```
  *
  * Datatype Invariant (DI):
  * - The sequence s can be empty or contain one or more integers, reflecting the
contents of the queue.
  * - The order of elements in s represents the order of elements in the
DoubleEndedQueue.
  *
  * Abstraction Function : AF(c) = s maps the concrete representation of the
DoubleEndedQueue to the abstract state s,
  *   capturing the sequence of integers from front to back.
  */
trait DoubleEndedQueue {

  /** Checks if the queue is empty.
    * Post: Returns true if s = [], otherwise false.
    *       The state s remains unchanged.
    */
  def isEmpty: Boolean

  /** Adds an integer x to the start (left end) of the queue.
    * Post: s' = [x] + s
    */
  def addLeft(x: Int): Unit

  /** Removes and returns the integer at the start (left end) of the queue.
    * Pre: s ≠ []
    * Post: If s = [x] + s', then s = s' and returns x.
    */
  def getLeft(): Int

  /** Adds an integer x to the end (right end) of the queue.
    * Post: s' = s + [x]
    */
  def addRight(x: Int): Unit

  /** Removes and returns the integer at the end (right end) of the queue.
    * Pre: s ≠ []
    * Post: If s = s' + [x], then s = s' and returns x.
    */
  def getRight(): Int
}
```

Here is the abstract state - written as a Scala `trait`.

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

class DoubleEndedQueue {
  private var head: Node = null
```

```
  private var tail: Node = null

  /** isEmpty
    * Post: returns True if the queue is empty, False otherwise
    */
  def isEmpty: Boolean = head == null

  /** addLeft
    * Pre: True
    * Post: s = x : s_0
    */
  def addLeft(x: Int): Unit = {
    val newNode = new Node(x, null, head)
    if (isEmpty) tail = newNode
    else head.prev = newNode
    head = newNode
  }

  /** getLeft
    * Pre: not (isEmpty)
    * Post: let s_0 = x : s' in s = s' ∧ returns x
    */
  def getLeft(): Int = {
    if (isEmpty) throw new NoSuchElementException("Queue is empty")
    val datum = head.datum
    head = head.next
    if (head != null) head.prev = null
    else tail = null
    datum
  }

  /** addRight
    * Pre: True
    * Post: s = s_0 ++ [x]
    */
  def addRight(x: Int): Unit = {
    val newNode = new Node(x, tail, null)
    if (isEmpty) head = newNode
    else tail.next = newNode
    tail = newNode
  }

  /** getRight
    * Pre: not (isEmpty)
    * Post: let s_0 = s' ++ [x] in s = s' ∧ returns x
    */
  def getRight(): Int = {
    if (isEmpty) throw new NoSuchElementException("Queue is empty")
    val datum = tail.datum
```

```
    tail = tail.prev
    if (tail != null) tail.next = null
    else head = null
    datum
  }
}
```