QUESTION 2

(a)

```
trait ReversableBuffer {
  // State : B = list of integers
  // init :  B = {}

  // add x to the end of this buffer
  // Post :  B = B0 ++ [x]
  def append (x: int)

  // add x to the start of this buffer
  // Post:   B = x : B0
  def prepend (x: int)

  // remove and return the i-th element, counting from zero
  // Pre :  length B0 ≥ i
  // Post:  return (B0 !! i) and B = take i B0 ++ drop (i+1) B0
  def get ( i : int) : int

  // reverse the contents of this buffer
  // Post:  B = reverse B0
  def rev
}
```

(b)

```
// Representing a buffer using a doubly-linked list with dummy headers (head and end)
// Let L(a,b) = [] if a = b ;  a : L(a.next, b), otherwise, and L(a) = L(a, null)

// Abstraction function: buffer = { b.datum | b is in init (L (buffer.next)) }
// DTi :  init(L(buffer.next)) is finite

class LLBuffer extends ReversableBuffer {
    private var buffer = new LLBuffer.Node (0, null, null)
    private var end  = new LLBuffer.Node (0, null, null)
    buffer.next = end
    end.prev = buffer
```

```
// We used a doubly-linked list with dummy headers to have append and prepend in O(1)
def append (x: int) = {
    var n1 = new LLBuffer.Node (x, end.prev, end)
    end.prev.next = n1
    end.prev = n1
}
def prepend (x: int) = {
    var n1 = new LLBuffer.Node (x, buffer, buffer.next)
    buffer.next.prev = n1
    buffer.next = n1
}
// Pre: the length of the list is at least i (size ≥ i)
// O(size)
def get (i : int) : int = {
    var j = 0
    var current = buffer.next
    // invariant : current.datum is the j-th element of the list
    while (i > j)
    {
      current = current.next
      j += 1
    }
    // i == j => we return current.datum and we remove the current node from the list
    var result = current.datum
    current.prev.next = current.next
    current.next.prev = current.prev
    result
}
// We swap the first element with the last one, then the next ones and so on
// O(size)
def rev = {
    var start = buffer.next
    var stop = end.prev
    // L(start, end) is unchanged ← invariant, but the rest of the list is in the right order
    if (start.prev != stop)  // happens when the list is non-empty
```

```
    while (start != stop)
    {
      //We have 2 cases, either start.next = stop, or they have at least one node between them
      var aux = start.datum
      start.datum = stop.datum
      stop.datum = aux
      start = start.next
      stop = stop.prev
      if (start.prev == stop) // if they were adjacent
          start = stop // so that we exit the loop
      //else we keep going
    }
}
}

//Companion object
object LLBuffer{
    private class Node(var datum : int, var prev: Node, var next: Node)
}
```