CS 410 Summer 2000
Question 1 for Homeowrk 4
Due 11:30 AM, Monday July 24

1. Before we get started with the setup for this problem, let's define a little notation. We represent strings on paper using angle brackets $\langle$ and $\rangle$, and the string is a comma separated sequence of items. Thus the string $\langle a, b, c\rangle$ is a string with 3 items, the string $\langle\langle a, b\rangle, a\rangle$ is a string with two items, the first of which is a string with 2 items. Thus $\langle\rangle$ is an empty string. We'll overload the + operator to mean string concatenation. Remember, these are mathematical strings, not programming language strings.

   Lets look at an abstraction for a OneWayList, defined below by a *contract.* This contract represents the conditions each procedure requires before it starts, and if those conditions are true, details what conditions will be true after the procedure finishes. In a contract programming model, requirements don't need to be tested within the procedures, only by the user. The procedures require that input will be a certain way; it is up to the calling routines to ensure this end of the contract (the contract procedures ensure the "ensures" part of the contract). The OneWayList is conceptually a string of items (items being an abstract type; in C++ we'd use a template; in Java we may write this using Objects).

   We have a single access point to this string, and rather than representing the access point with an index, it's simpler to write the contracts by considering the string to be in two pieces: a *left* and a *right.* Our access point is the first item of the *right* string. (We'll overload the math symbol + to indicate string concatenation). The string represented by the OneWayList is then the concatenation $\textit{left} + \textit{right}$. (We consider the OneWayList to be a string because a string is a mathematical object with known properties and some simple theorems, while a linked list is a computer science representation. In order to formulate mathematical constraints, we need a mathematical representation.) We also have the following operations:

   `void addRight (item i)`
   requires: none
   ensures: $\textit{left} = \#\textit{left}, \textit{right} = \#i + \#\textit{right}$

   To explain the notation: a # before a variable name indicates the value of that variable before the operation began, and an unadorned variable name indicates the value of that variable after the operation. So the contract above says that the left string is unmodified, and the right string is now the item passed in, concatenated with the old right string.

   `item removeRight ()`
   requires: $\textit{right} \neq \langle\rangle$
   ensures: $\textit{left} = \#\textit{left}, \#\textit{right} = \texttt{removeRight} + \textit{right}$

   `item peek ()`
   requires: $\textit{right} \neq \langle\rangle$
   ensures: $\textit{left} = \#\textit{left}$ and $\textit{right} = \#\textit{right}$ and $\exists$ item t, string of item s such that $\texttt{peek} = t$ and $\textit{right} = t + s$

`void advance ()`
requires: $\mathit{right} \neq \langle\rangle$
ensures: $\exists$ item t such that $\mathit{left} = \#\mathit{left} + t$ and $t + \mathit{right} = \#\mathit{right}$

`void moveToStart ()`
requires: none
ensures: $\mathit{left} = \langle\rangle$, $\mathit{right} = \#\mathit{left} + \#\mathit{right}$

`void moveToFinish ()`
requires: none
ensures: $\mathit{left} = \#\mathit{left} + \#\mathit{right}$, $\mathit{right} = \langle\rangle$

`int leftLength ()`
requires: none
ensures: leftLength $= |\mathit{left}|$ and $\mathit{left} = \#\mathit{left}$ and $\mathit{right} = \#\mathit{right}$

`int rightLength ()`
requires: none
ensures: rightLength $= |\mathit{right}|$ and $\mathit{left} = \#\mathit{left}$ and $\mathit{right} = \#\mathit{right}$

Note that some of these operations are not strictly necessary, as they can be implemented using other operations. The basic rule of thumb in designing an abstraction is to limit the primary operations to the smallest possible set that still allows full functionality. An additional rule is that if a secondary operation (one implemented using primary operations) will be more efficient as a primary operation, it should be so. Thus the reason for `moveToFinish` as a primary operation. `peek` is listed as a primary operation for convenience.

Also note that if any procedure has a requires clause, the condition specified must be either testable by the user, or the abstraction designer must provide a way to test the condition. Thus the need for a `rightLength` function; without it we couldn't tell if $\mathit{right} \neq \langle\rangle$.

We should also specify a constructor and destructor:

`OneWayList ()`
ensures: $\mathit{left} = \langle \mathsf{nil} \rangle$, $\mathit{right} = \langle \mathsf{nil} \rangle$

`~OneWayList ()`
requires: none

Note that the constructor has no requires, and the destructor has no ensures. This is because the abstraction has no existence before the constructor is called, nor after the destructor is called.

One way to implement this abstraction is as a linked list. Consider the following data definition:

```
class Node {
  item data;
  Node next;
}

class OneWayList {
  Node prefront;
```

```
    Node current;
    Node last;
    int left_length;
    int right_length;
    operations as above
  }
```

The **prefront** node is a dummy node to eliminate many special cases in the implementation. It is always the first thing in the linked list. Here's some code for some of the operations to get you started on how this particular implementation is supposed to work:

```
  OneWayList () {
    prefront = new Node();
    prefront.next = NULL;
    current = prefront;
    last = prefront;
    left_length = right_length = 0;
  }


  void peek () {
    return current.next.data;
  }


  void advance () {
    current = current.next;
    left_length++;
    right_length--;
  }


  void moveToStart () {
    current = prefront;
    right_length += left_length;
    left_length = 0;
  }
```

Note carefully that the `current` pointer in the concrete representation points to the item *before* the current one. This is so that `removeRight` is an $O(1)$ operation. Also note that we're not checking if `current.next` is NULL before dereferencing; because we're in a contract programming model, we may assume that everything in the requires clause is true.

(a) Write code for the `addRight` method and the `removeRight` method.

(b) Write a contract for a procedure `quickSort (OneWayList l)`, and implement it using only the provided OneWayList operations. Do not assume anything about the implementation of the OneWayList. You may assume that the type `item` has an operator $<$. (In C++, we could overload one; in Java we could use a `before` method as in homework 2). Hint: remove the pivot from the list first, and do not put the pivot into either partition, but rather place it between the partitions after the recursive calls. The partition function should be rewritten entirely to build two new lists.

(c) Do a run-time analysis of your *quickSort*.