Exercises

1. Fill in the blanks in the initialization of `node3`, `node2`, `node1` and `head`, so that `node1`, `node2`, and `node3` form a linked list (in that order) referred to by `head`.

```
ListNode node3 = new ListNode("Node 3",_________ );
ListNode node2 = new ListNode("Node 2",_________ );
ListNode node1 = new ListNode("Node 1",_________ );
ListNode head = _________ ;
```

2. Fill in the blanks in the following method:

```
// Returns true if the list referred to by head
// has at least two nodes; otherwise returns false
public boolean hasTwo(ListNode head)
{
  return ___________________________________________ ;
}
```

3. Write a method

```
public ListNode removeFirst(ListNode head)
```

that unlinks the first node from the list and returns the head of the new list. Your method should throw a `NoSuchElementException` when the original list is empty.

4. Write a method

```
public int size(ListNode head)
```

that returns the number of nodes in the list referred to by `head`:

(a) using a `for` loop
(b) using recursion.

5. `head` is the fist node of a non-empty linked list (without a tail). Write a method

```
public void add(ListNode head, Object value)
```

that appends a new node holding `value` at the end of the list.

6. Fill in the blanks in the method below. This method takes the list referred to by `head`, builds a new list in which the nodes have the same information but are arranged in reverse order, and returns the head of the new list. The original list remains unchanged. Your solution must use a `for` loop (not recursion).

```
public ListNode reverseList(ListNode head)
{
  ListNode node, newNode, newHead = null;

  for ( _____________________________________________ )
  {
      _____________________________________________
      ...
  }

  return newHead;
}
```

7. Write a method

```
public ListNode concatenateStrings(ListNode head)
```

that takes the list referred to by `head`, builds a new list, and returns its head. The original list contains strings. The *k*-th node in the new list should contain the concatenation of all the strings from the original list from the first node up to and including the *k*-th node. For example, if the original list contains strings `"A"`, `"B"`, `"C"`, the new list should contain strings `"A"`, `"AB"`, `"ABC"`.

8. Write a method

```
public ListNode rotate(ListNode head)
```

that takes a linked list referred to by `head`, splits off the first node, and appends it at the end of the list. The method should accomplish this solely by rearranging links: do not allocate new nodes or move objects between nodes. The method should return the head of the rotated list.

9. A list referred to by `head` contains strings arranged alphabetically in ascending order. Write a method

```
public ListNode insertInOrder(ListNode head, String s)
```

that inserts `s` into the list, preserving the order. If `s` is already in the list, it is not inserted. The method should return the head of the updated list.

10. Write a method

```
public ListNode middleNode(ListNode head)
```

that returns the middle node (or one of the two middle nodes) of a linked list. Design this method using no recursion and only one loop.

11.◆ Let us say that a string matches an input pattern (another string) if the pattern is at least as long as the string and for every non-wildcard character in the pattern the string has the same character in the same position. (The wildcard character is `'?'`.) For example, both `"New York"` and `"New Jersey"` match the input pattern `"New ???????"`. Write a method

```
public ListNode moveToBack(ListNode head, String pattern)
```

that takes a list of strings referred to by `head` and moves all the strings that match `pattern` to the end of the list, preserving their order. Your method must work by rearranging links; do not allocate new nodes or use temporary arrays or lists. The method should return the head of the updated list.

12. Write a method

```
public void concat(LinkedListWithTail list1,
                   LinkedListWithTail list2)
```

that concatenates `list2` to `list1` in $O(1)$ time.

13. Implement Selection Sort for a linked list with a tail. (Assume that the list holds `Comparable` objects.) You method should run in $O(n^2)$ time, where n is the number of nodes in the list.

14. Write the following method:

```
// Removes the largest element from the list and returns
// the head of the modified list.
// Precondition: head refers to the first node of a
//               non-empty doubly-linked list with
//               Integer values.
public ListNode2 removeMax(ListNode2 head)
```

15.◆ Write a method

```
public void quickSort(ListNode2 fromNode, ListNode2 toNode)
```

that implements the Quicksort algorithm for a given segment of a doubly-linked list. `fromNode` refers to the first node of the segment, and `toNode` refers to the last node of the segment. (Assume that the list holds `Comparable` objects.) The links outside the segment should remain unchanged, and the segment should remain linked at the same place within the list. The method should run in $O(n \log n)$ time, where n is the number of nodes in the segment. Do not use any temporary arrays or lists.