Practice Exercise #29: Self-Adjusting List

http://www.comp.nus.edu.sg/~cs1020/4_misc/practice.html

Objective:

- Programming on linked list

Task Statement

A *self-adjusting* data structure is one where it can rearrange its elements to improve efficiency.

We know that searching for an item in a linked list is slow, as we need to conduct a sequential search from the first node of the list. Suppose we are given many search operations. If we take the assumption that a searched item is likely to be searched again in the near future (this is known as the principle of *temporal locality*), then it pays to move the searched item to the front of the list, so that the next time it is searched, it can be found more quickly.

For each search operation, we can define the number of "probes" to be the number of elements in the list it needs to examine before it finds the search item or before we conclude that the item is not in the list.

With a self-adjusting list, the average number of probes would be reduced compared to a list without such self-adjusting ability.

Most of the code is already given in the skeleton programs. To keep it simple, we implement **MyLinkedList** class as a standalone class instead of it being an implementation of an interface as shown in lecture. Also, to keep the number of files small, we provide two skeleton programs: **MyLinkedList.java** contains the definition of both **ListNode** and **MyLinkedList** classes, and **SelfAdjustList.java** contains the client program.

You are not to modify **SelfAdjustList.java** (and hence you do not need to submit it). You are to complete the **toString()** and **search()** methods in **MyLinkedList** class. The **search()** method is to compute the number of probes needed for a search operation.

You are not to modify the rest of the given code in **MyLinkedList**. You need to submit only **MyLinkedList.java**.

Sample run:

Inputs are shown in blue.

```
5
Avery Bubble Candy Diana Elaine
Original list:
[Elaine, Diana, Candy, Bubble, Avery]
6
Avery Candy Zillion Candy Avery Candy
Average number of probes = 3.17
Final list:
[Candy, Avery, Elaine, Diana, Bubble]
```

Explanation:

First, the client program reads in 5 names and sets up the list:

Elaine → Diana → Candy → Bubble → Avery

There are 6 search operations:

1st: Search for "Avery" requires 5 probes, "Avery" is moved to the front with resulting list:

Avery → Elaine → Diana → Candy → Bubble

2nd: Search for "Candy" requires 4 probes, "Candy" is moved to the front with resulting list:

Candy → Avery → Elaine → Diana → Bubble

3rd: Search for "Zillion" requires 5 probes, item not found so list remains unchanged:

Candy → Avery → Elaine → Diana → Bubble

4th: Search for "Candy" requires 1 probe, list remains unchanged:

Candy → Avery → Elaine → Diana → Bubble

5th: Search for "Avery" requires 2 probes, "Avery" is moved to the front with resulting list:

Avery → Candy → Elaine → Diana → Bubble

6th: Search for "Candy" requires 2 probes, "Candy" is moved to the front with resulting list:

Candy → Avery → Elaine → Diana → Bubble

Total number of probes = 5 + 4 + 5 + 1 + 2 + 2 = 19. Hence, average probes per search = 19/6 or 3.17.

Compare this with a non-self-adjusting list which will give a total of 24 probes, or an average of 4.0 probes per search operation.