Questions

1. Consider the following Java code for a singly linked list method `getIndexOf(E e)` that returns the index of the first occurrence of given element in a singly linked list if that element is present, and it returns -1 if the element is not present. For example, if the given argument `e` is at the front of the list then the method returns index 0.

You may assume the list is not empty, and the field `head` is defined in the usual way.

*Provide the missing Java code of this method below.*

```java
public int getIndexOf(E e){
    SNode<E> cur = head;
    int index = 0;

    // ADD YOUR CODE HERE

    if (cur.element == e)
        return index;
    else
        return -1;
}
```

2. Fill in the code of the following Java method `reverse()` which reverses the order of nodes in a singly linked list, by changing `next` references so that they go in the opposite direction in the list. The `head` and `tail` references must be swapped too.

The idea of the solution below is to iterate from the `head` node to the `tail` node. While doing so, partition of the nodes into two lists: the (reversed) nodes up to the current node, and the (not yet reversed) nodes beyond the current node. The heads of the two lists are `headList1` and `headList2`.

*To answer such a question, many people find it helpful to visualize the situation with boxes (nodes) and arrows, as done in the lectures. Doing it in your head is likely to be too difficult.*

```java
public void reverse(){
    SNode<E> headList1, headList2, cur;
    if (head != null){
        headList1 = null;
        headList2 = head;

        // ADD YOUR CODE HERE. In formulating your solution, try to focus
        // first on the general case, rather than the two edge cases.

        tail = head;
        head = headList1;  // the tail of the original list
    }
}
```
3. [EDITED Nov 19. After presenting lecture 29 on interfaces, I went back and reviewed the Iterator methods for the singly linked list class that I had presented earlier, and I rewrote it. The link below is to the solution code only. I am figuring that at this point in the semester, if you haven’t done this exercise by now then you probably aren’t going to do it at all. Next time I teach the course, I will add stubs mentioned below. In the meantime, enjoy reading the solution code.]

See the Java code

http://www.cim.mcgill.ca/~langer/250/MyLinkedLists.zip

which contains a partially implemented singly linked list class. The java files with "stubs" in the file names contains some partially implemented methods:

- remove(i)
- add(i,e)
- reverse() i.e. question 2 above

Your task is to implement these methods. A solution file is provided.

Note: Assignment 1 will use linked lists, but it does not require you to practice the "under the hood" manipulations of linked lists. These exercises give you an opportunity for doing so.

4. The URL in the previous question also contains a partially implemented doubly linked list class (DLinkedList_stubs.java) along with a fulling implemented class.

Implement the following methods in the (DLinkedList_stubs.java) class:

- addBefore(E e, DNode node)
  This is a helper method that is used by various add methods, namely addFirst(), addLast(), add()
- reverse()
  i.e. same as in Questions 2 and 3, but now with a doubly linked list.

5. Consider the Java code:

```java
public void display( LinkedList<E> list ){
    for (int i = 0; i < list.size(); i++){
        System.out.println( list.get(i).toString() );
    }
}
```

How does the number of steps of this method depend on \(N\), the number of elements in the list. Unlike in the example in lecture 5, consider the fact that the get(i) method will start from the tail of the list in the case that i is greater than \(N/2\).

6. Can you have a loop in a singly linked list? That is, if you follow the next references, then can you reach a node that you have already visited (and hence loop around infinitely many times if you keep advancing by following the next reference?)
Solutions

1. The solution is to advance by following the next pointers. However, be careful not to advance past the last element of the list because you need to call `cur.getElement` after the code you add.

   ```java
   while ((cur.element != e) && (cur.next != null)) {
       cur = cur.next;
       index++;
   }
   ```

2. In the general case, `headList1` is pointing to the node which came before it in the original list (since list 1 had already been reversed) and `headList2` is pointing to the node that came after it in the original list (since list 2 hasn't been reversed yet). The goal is to remove the first element from list 2 and add it to list 1. We need a temporary variable to help us, and we can use `cur` which has been declared.

   ```java
   while (headList2 != null) {
       // the condition here handles an edge case:
       // namely we're done when list 2 is empty
       cur = headList2.next;
       headList2.next = headList1;
       headList1 = headList2;
       headList2 = cur;
   }
   ```

3. See the implemented method in the `SLinkedList1` class which is part of the zip file that I have provided in the question.

4. See the implemented methods in the `DLinkedList.java` file.

5. For any index $i$ the first half of the list, it takes $i$ steps to get to the node. So the number of steps total for nodes in the first half of the list is:

   $$(1 + 2 + 3 + ... + \frac{N}{2}) = \frac{\frac{N}{2} (\frac{N}{2} + 1)}{2}$$

   For nodes in the second half of the list, we start from the tail instead of the head, but the idea is the same, so it takes

   $$(1 + 2 + 3 + ... + \frac{N}{2}) = \frac{\frac{N}{2} (\frac{N}{2} + 1)}{2}$$

   steps in total to reach those nodes. Thus, in total the number of steps is the sum of the above, or

   $\frac{N}{2} \left(\frac{N}{2} + 1\right)$.

   This is about twice as fast as using the inefficient `getNode()` method, but it is still $O(N^2)$. 

last updated: 19th Nov, 2016 at 09:23
6. Linked list data structures do allow for a loop, in the sense that there is nothing stopping the \texttt{next} field of some node from referencing a node earlier in the list. However, in this case, the data structure will not be a "list", in the sense of having a well defined ordering from 0, 1, ..., size -1. If the linked list class is properly implemented, then the methods should not allow this to happen.