Questions

Questions 1-3 are on singly linked lists. Questions 4-5 are on doubly linked lists.

1. See the file `Exercises_LinkedList_JavaCode.zip`. You will need to put this code into the correct packages. The `stub` code contains the questions. The non-`stub` code contains the solutions.

   (a) Fill in the missing code of the following methods in the `SLinkedList_stub.java` class :

   - add(int i, E element)
   - getIndexOf(E e)
   - remove(int i)
   - getNode(int i)

   (b) (More challenging) Fill in the code of the method `reverse()` which reverses the order of elements in a *singly linked list*. The idea of the method is to reverse the order of the nodes (not reversing the elements), by changing `next` references so that they go in the opposite direction in the list. The `head` and `tail` references must be swapped too. To do so, iterate from the `head` node to the `tail` node. At each step, partition the list into two sub-lists: (1) the (reversed) nodes up to the current node and (2) the not-yet-reversed nodes beyond the current node. The heads of the two lists are `headList1` and `headList2`.

   You may find it helpful to visualize the linked list by drawing boxes (nodes) and arrows, as done in the lectures. Doing it in your head is likely to be too difficult.

2. 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) ?

   (This is a pseudocode question.)

3. Suppose you have a reference to a node in a singly linked list and this node is not the last one in the list.

   (a) How could you remove the element at this node from the list, while maintaining a proper linked list data structure? Note that this would reduce the number of nodes by 1. Your solution should not require looking for this node by starting at the head of the list, but rather you must do it in constant time i.e. O(1). The solution is just a few lines of code. Don't peek!

   (b) How could you insert an element into the list at the position that comes *before* the element at this given node?

   (This is a pseudocode question.)

4. Implement the following methods in the `DLinkedList_stubs.java` class:

   - `remove(int i)`

     This method first calls `getNode(int i)` which returns a reference to a node. `getNode(int i)` was discussed in the lecture. `remove(i)` then removes this node from the list, and this is the part you need to implement.

   - `addBefore(E e, DNode<E> node)`

     This is a private helper method which is called by various add methods.

   - `reverse()`

     i.e. same as in Question 1 but now with a doubly linked list.

5. Consider the Java code:

```
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 `size`, the number of elements in the list?

   (a) Consider the case that the `get(i)` method starts at the front of the list.

   (b) Consider the case that the `get(i)` method will start from the tail of the list in the case that i is greater than `size`/2.

   Part (a) was discussed in the lecture, but not part (b).

Solutions

1. The solution code is found in the `SLinkedList.java` file.

   For the `reverse()` method, see the figures and description here:
   `http://www.cim.mcgill.ca/~langer/250/E3-slinkedlist-reverse.pdf`

2. Linked list data structures do allow for a loop, in the sense that there is nothing stopping the `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.

3. (a) Let `cur` be the reference to the given node.

```
cur.element = cur.next.element   //   Copy the element at the next node
//   back to the current one.
cur.next = cur.next.next         //   Skip over the next node.
```

   This reduces the size of the list by 1 and removes the element that had been at the current node. The node that was removed still references the next node. That's not a problem since nothing references that node and so eventually it will be taken away by the garbage collector. Alternatively, you could insert the following in a suitable place (see comment).

```
tmp      = cur.next     //  Insert after the first instruction above.
tmp.next = null         //  Insert after the second instruction above.
```

   (b) Here the idea is similar. We insert a node after the current node and let that node's next field point to the same node as `cur.next`

```
tmp  = new node
tmp.next = cur.next
```

   Then, change the next field of the current node to point to the new node. Finally, move the elements to their appropriate nodes.

```
cur.next = tmp
tmp.element = cur.element
cur.element = new element // the one to be added
```

4. The solution is found in the `DLinkedList.java` file.

5. (a) If the `get(i)` method always starts at the front of the list, then it requires `i` steps to reach node `i`. So the total number of steps when calling `get(i)` for `i` in 0 to N-1 is

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

   which is $O(N^2)$. You might ask whether the sum instead should be

$$(0 + 1 + 2 + 3 + ... + N - 1) \;=\; \frac{N(N-1)}{2}$$

Yes, that would be fine. It is also $O(N^2)$ which is the main point here.

(b) 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

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

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

