Problem Set 3 - Solution

Linked Lists, Recursion

---

1. *Given the following definition of a circular linked list (CLL) class:

```
public class Node {
   public String data;
   public Node next;
   public Node(String data, Node next) {
       this.data = data; this.next = next;
   }
}

public class LinkedList {
   private Node rear;  // pointer to last node of CLL
   ...
}
```

The class keeps a circular linked list, with a `rear` pointer to the last node.

Implement the following method in the `LinkedList` class, to delete the *first* occurrence of a given item from the linked list. The method returns true if the item is deleted, or false if the item is not found.

```
public boolean delete(String target) {
     /* COMPLETE THIS METHOD */
}
```

SOLUTION

```
public boolean delete(String target) {

   if (rear == null) { // list is empty
       return false;
   }

   if (rear == rear.next) { // list has only one node
      if (target.equals(rear.data)) { // found, delete, leaves empty list
         rear = null;
         return true;
      } else {   // not found
         return false;
      }
   }

   Node prev=rear, curr=rear.next;
   do {
       if (target.equals(curr.data)) {
            prev.next = curr.next;
            if (curr == rear) { // if curr is last node, prev becomes new last node
```

```
                rear == prev;
            }
            return true;
        }
        // skip to next node
        prev = curr;
        curr = curr.next;
    } while (prev != rear);
    return false; // not found
}
```

---

2. * Implement a method in the circular linked list class of problem 1, to add a new item *after* a specified item. (If the new item happens to be added after the last, then it should become the new last item.) If the item does not exist in the list, the method should return false, otherwise true.

```
public boolean addAfter(String newItem, String afterItem)
throws NoSuchElementException {
     /* COMPLETE THIS METHOD */
}
```

SOLUTION

```
public boolean addAfter(String newItem, String afterItem)
throws NoSuchElementException {
    if (rear == null) { // empty
       return false;
    }
    Node ptr=rear;
    do {
        if (afterItem.equals(ptr.data)) {
           Node temp = new Node(newItem,ptr.next);
           ptr.next = temp;
           if (ptr == rear) { // new node becomes last
               rear = temp;
           }
           return true;
        }
        ptr = ptr.next;
    } while (ptr != rear);
    return false;   // afterItem not in list
 }
```

---

3. A *doubly linked list* (DLL) is a linked list with nodes that point both forward and backward. Here's an example:

```
3 <---> 5 <---> 7 <---> 1
```

Here's a DLL node definition:

```
public class DLLNode {
   public String data;
   public DLLNode prev, next;
   public DLLNode(String data, DLLNode next, DLLNode prev) {
       this.data = data; this.next = next; this.prev = prev;
```

```
    }
}
```

The `next` of the last node will be null, and the `prev` of the first node will be null.

Implement a method to move a node (given a pointer to it) to the front of a DLL.

```
// moves target to front of DLL
public static DLLNode moveToFront(DLLNode front, DLLNode target) {
      /** COMPLETE THIS METHOD **/
}
```

SOLUTION

```
// moves target to front of DLL
public static DLLNode moveToFront(DLLNode front, DLLNode target) {
    if (target == null || front == null || target == front) {
       return;
    }
    // delink the target from the list
    target.prev.next = target.next;
    // make sure there is something after target before setting its prev
    if (target.next != null) {
       target.next.prev = target.prev;
    }
    target.next = front;
    target.prev = null;
    front.prev = target;
    return target;
}
```

---

4. With the same `DLLNode` definition as in the previous problem, implement a method to reverse the sequence of items in a DLL. Your code should NOT create any new nodes - it should simply resequence the original nodes. The method should return the front of the resulting list.

```
public static DLLNode reverse(DLLNode front) {
      /** COMPLETE THIS METHOD **/
}
```

SOLUTION

```
public static DLLNode reverse(DLLNode front) {
    if (front == null) {
       return null;
    }
    DLLNode rear=front, prev=null;
    while (rear != null) {
       DLLNode temp = rear.next;
       rear.next = rear.prev;
       rear.prev = temp;
       prev = rear;
       rear = temp;
    }
    return prev;
}
```

---

5. Implement a RECURSIVE method to delete all occurrences of an item from a (non-circular) linked list. Use the Node class definition of problem 1. Return a pointer to the first node in the updated list.

```
public static Node deleteAll(Node front, String target) {
     /* COMPLETE THIS METHOD */
}
```

SOLUTION

```
public static Node deleteAll(Node front, String target) {
    if (front == null) { return null; }
    if (front.data.equals(target)) {
       return deleteAll(front.next, target);
    }
    front.next = deleteAll(front.next, target);
    return front;
}
```

---

6. * Implement a RECURSIVE method to merge two **sorted** linked lists into a single **sorted** linked list WITHOUT duplicates. No new nodes must be created: the nodes in the result list are a subset of the nodes in the original lists, rearranged appropriately. You may assume that the original lists do not have any duplicate items.

   For instance:

   ```
   l1 = 3->9->12->15
   l2 = 2->3->6->12
   ```

   should result in the following:

   ```
   2->3->6->9->12->15
   ```

   Assuming a **Node** class defined like this:

   ```
   public class Node {
      public int data;
      public Node next;
   }
   ```

   Complete the following method:

   ```
   public static Node merge(Node frontL1, Node frontL2) {
      ...
   }
   ```

   SOLUTION

   ```
   public static Node merge(Node frontL1, Node frontL2) {
        if (frontL1 == null) { return front L2; }
        if (frontL2 == null) { return front L1; }
        if (frontL1.data == frontL2.data) {
           // keep one copy
           frontL1.next = merge(front1.next, frontL2.next);
   ```

```
            return frontL1;
        }
        if (frontL1.data < frontL2.data) {
            frontL1.next = merge(front1.next, frontL2);
            return frontL1;
        }
        frontL2.next = merge(front2.next, frontL1);
        return frontL2;
    }
```
