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

```
2.       public class Node {
3.          public String data;
4.          public Node next;
5.          public Node(String data, Node next) {
6.              this.data = data; this.next = next;
7.          }
8.       }
9.
10.       public class LinkedList {
11.          private Node rear;  // pointer to last
  node of CLL
12.          ...
13.       }
14.
```


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.


```
15.      public boolean delete(String target) {
16.           /* COMPLETE THIS METHOD */
17.      }
18.
19.
```

SOLUTION

```
  public boolean delete(String target) {
20.
21.    if (rear == null) { // list is empty
22.        return false;
23.    }
24.
25.    if (rear == rear.next) { // list has only one
  node
26.       if (target.equals(rear.data)) { // found,
  delete, leaves empty list
27.          rear = null;
```

```
28.          return true;
29.       } else {   // not found
30.          return false;
31.       }
32.    }
33.
34.    Node prev=rear, curr=rear.next;
35.    do {
36.        if (target.equals(curr.data)) {
37.              prev.next = curr.next;
38.              if (curr == rear) { // if curr is last
   node, prev becomes new last node
39.                  rear == prev;
40.              }
41.              return true;
42.        }
43.        // skip to next node
44.        prev = curr;
45.        curr = curr.next;
46.    } while (prev != rear);
47.    return false; // not found
48.}
49.
```


50.* 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.

```
51.     public boolean addAfter(String newItem, String
   afterItem) {
52.          /* COMPLETE THIS METHOD */
53.     }
54.
55.
```

SOLUTION

```
56.      public boolean addAfter(String newItem,
  String afterItem) {
57.         if (rear == null) { // empty
58.            return false;
59.         }
60.         Node ptr=rear;
61.         do {
62.             if (afterItem.equals(ptr.data)) {
63.                Node temp = new
  Node(newItem,ptr.next);
64.                ptr.next = temp;
65.                if (ptr == rear) { // new node
  becomes last
66.                    rear = temp;
67.                }
68.                return true;
69.             }
70.             ptr = ptr.next;
71.         } while (ptr != rear);
72.         return false;   // afterItem not in list
73.      }
74.
```


75. 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
76.
Here's a DLL node definition:

```
77.      public class DLLNode {
78.         public String data;
79.         public DLLNode prev, next;
80.         public DLLNode(String data, DLLNode next,
  DLLNode prev) {
81.             this.data = data; this.next = next;
  this.prev = prev;
82.         }
83.      }
```

84. 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
85.    public static DLLNode moveToFront(DLLNode front, DLLNode target) {
86.          /** COMPLETE THIS METHOD **/
87.    }
```

88. **SOLUTION**

```
      // moves target to front of DLL
89.    public static DLLNode moveToFront(DLLNode front, DLLNode target) {
90.        if (target == null || front == null || target == front) {
91.          return;
92.        }
93.        // delink the target from the list
94.        target.prev.next = target.next;
95.        // make sure there is something after target before setting its prev
96.        if (target.next != null) {
97.          target.next.prev = target.prev;
98.        }
99.        target.next = front;
100.        target.prev = null;
101.        front.prev = target;
102.        return target;
103.    }
```

104.

105. 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) {
106.            /** COMPLETE THIS METHOD **/
107.      }
108.
```

SOLUTION

```
      public static DLLNode reverse(DLLNode front) {
109.          if (front == null) {
110.             return null;
111.          }
112.          DLLNode rear=front, prev=null;
113.          while (rear != null) {
114.             DLLNode temp = rear.next;
115.             rear.next = rear.prev;
116.             rear.prev = temp;
117.             prev = rear;
118.             rear = temp;
119.          }
120.          return prev;
121.      }
122.
```


123. 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.

```
124.    public static Node deleteAll(Node front,
    String target) {
125.          /* COMPLETE THIS METHOD */
126.    }
127.
```

SOLUTION

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

```
131.                return deleteAll(front.next,
  target);
132.            }
133.            front.next = deleteAll(front.next,
  target);
134.            return front;
135.        }
136.
```

137.* 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
138.   l2 = 2->3->6->12
```


139. should result in the following: `2->3->6->9->12->15`

140. Assuming a Node class defined like this:


```
      public class Node {
141.        public int data;
142.        public Node next;
143.    }
```

144. Complete the following method:

```
145.    public static Node merge(Node frontL1, Node
  frontL2) {
146.        ...
147.    }
```

148. **SOLUTION**

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

```
149.            if (frontL1 == null) { return front L2;
  }
150.            if (frontL2 == null) { return front L1;
  }
151.            if (frontL1.data == frontL2.data) {
152.               // keep one copy
153.               frontL1.next = merge(front1.next,
  frontL2.next);
154.               return frontL1;
155.            }
156.            if (frontL1.data < frontL2.data) {
157.               frontL1.next = merge(front1.next,
  frontL2);
158.               return frontL1;
159.            }
160.            frontL2.next = merge(front2.next,
  frontL1);
161.            return frontL2;
162.      }
```