1. Assuming an **IntNode** class defined like this:

```
2.       public class IntNode {
3.          public int data;
4.          public IntNode next;
5.          public IntNode(int data, IntNode next) {
6.              this.data = data; this.next = next;
7.          }
8.          public String toString() {
9.              return data + "";
10.      }
11.
```

Implement a method that will add a new integer before a target integer in the list. The method should return a pointer/reference to the front node of the resulting list. If the target is not found, it should return front without doing anything:


```
12.         public static IntNode addBefore(IntNode
    front, int target, int newItem) {
13.             /* COMPLETE THIS METHOD */
14.         }
15.
```


SOLUTION

```
        public static IntNode addBefore(IntNode
    front, int target, int newItem) {
16.             IntNode prev=null, ptr=front;
17.             while (ptr != null && ptr.data !=
    target) {
18.                 prev = ptr;
19.                 ptr = ptr.next;
20.             }
21.             if (ptr == null) { // target not found
22.                 return front;
23.             }
24.             IntNode temp = Intnew Node(newItem,
    ptr); // next of new node should point to target
```

```
25.             if (prev == null) { // target is first
   item, so new node will be new front
26.                return temp;
27.             }
28.             prev.next = temp;
29.             return front;  // front is unchanged
30.         }
31.
32.
```

33. With the same **IntNode** class definition as above, implement a method that will add a new integer before the last item in a linked list. (In other words, the added integer will become the second-to-last item in the resulting linked list.) The method should return a pointer/reference to the front node of the resulting linked list. If the input linked list is empty, the method should return null, without doing anything.


```
34.         public static IntNode addBeforeLast(IntNode
   front, int item) {
35.         /* COMPLETE THIS METHOD */
36.         }
37.
```

SOLUTION

```
            public static IntNode addBeforeLast(IntNode
   front, int item) {
38.             if (front == null) {
39.                return null;
40.             }
41.             IntNode prev=null, ptr=front;
42.             while (ptr.next != null) {
43.                 prev = ptr;
44.                 ptr = ptr.next;
45.             }
46.
```

```
            IntNode temp = Intnew Node(item,
   ptr); // next of new node should point to last
            if (prev == null) { // added item is
   first, so new node will be new front
               return temp;
            }
            prev.next = temp;
            return front;  // front is unchanged
        }
```

Given the following definition of a `StringNode` class:

```
      public class StringNode {
         public String data;
         public StringNode next;
         public StringNode(String data, StringNode
   next) {
            this.data = data; this.next = next;
         }
         public String toString() {
            return data;
         }
      }
```


Implement a method that will search a given linked list for a target string, and return the number of occurrences of the target:

```
        public static int
   numberOfOccurrences(StringNode front, String
   target) {
              /* COMPLETE THIS METHOD */
        }
```

SOLUTION

```
73.       public static int
   numberOfOccurrences(StringNode front, String
   target) {
74.           int count=0;
75.           for (StringNode ptr=front;ptr !=
   null;ptr=ptr->next) {
76.               if (target.equals(ptr.data)) {
77.                  count++;
78.           }
79.           return count;
80.        }
81.
```

82.* Assuming the `IntNode` class definition of problem 1, implement a method to delete EVERY OTHER item from an integer linked list. For example:


```
83.    before: 3->9->12->21
84.     after: 3->12->21
85.
86.    before: 3->9->12->15
87.     after: 3->12
88.
89.    before: 3->9
90.     after: 3
91.
92.    before: 3
93.     after: 3
```

If the list is empty, the method should do nothing.

```
   public static void deleteEveryOther(IntNode front)
   {
94.              /* COMPLETE THIS METHOD */
95.        }
96.
```

SOLUTION

```
         public static void deleteEveryOther(IntNode
   front) {
97.            if (front == null) {
98.                return;
```

```
99.             }
100.             Node prev=front, ptr=front.next;
101.             boolean tbd=true;
102.             while (ptr != null) {
103.                 if (tbd) {
104.                     ptr = ptr.next;   // advance to
    after item to be deleted
105.                     prev.next = ptr;  // bypass
    item to be deleted
106.                     tbd = false;      // next item
    should not be deleted
107.                 } else {
108.                     prev = ptr;       // don't
    delete this (ptr) item, advance prev and ptr
109.                     ptr = ptr.next;
110.                     tbd = true;       // but mark
    next item for deletion
111.                 }
112.             }
113.         }
114.
```


115. * With the same `StringNode` definition as in the previous problem, implement a method that will delete all occurrences of a given target string from a linked list, and return a pointer to the first node of the resulting linked list:

```
116.         public static StringNode
    deleteAllOccurrences(StringNode front, String
    target) {
117.             /* COMPLETE THIS METHOD */
118.         }
119.
```

SOLUTION

```
    public static StringNode
    deleteAllOccurrences(StringNode front, String
    target) {
120.
121.   if (front == null) {
```

```
122.      return null;
123.   }
124.
125.   StringNode curr=front, prev=null;
126.
127.   while (curr != null) {
128.      if (curr.data.equals(target)) {
129.         if (prev == null) {       // target is the
   first element
130.            front = curr.next;
131.         } else {
132.            prev.next = curr.next;
133.         }
134.      } else {
135.         prev = curr;
136.      }
137.      curr = curr.next;
138.   }
139.
140.   return front;
141.}
142.
```


143.* Implement a (NON-RECURSIVE) method to find the common elements in two **sorted** linked lists, and return the common elements in **sorted** order in a NEW linked list. The original linked lists **should not** be modified. So, for instance, `l1 = 3->9->12->15->21`

144. `l2 = 2->3->6->12->19`

145.

should produce a new linked list: `3->12`

146.

You may assume that the original lists do not have any duplicate items.Assuming an **IntNode** class defined like this:

```
147.      public class IntNode {
148.         public int data;
```

```
149.         public IntNode next;
150.         public IntNode(int data, IntNode next) {
151.             this.data = data; this.next = next;
152.         }
153.         public String toString() {
154.             return data + "";
155.     }
156.
```

Complete the following method:

```
    // creates a new linked list consisting of the items common to the input lists
157.     // returns the front of this new linked list, null if there are no common items
158.     public IntNode commonElements(IntNode frontL1, IntNode frontL2) {
159.         ...
160.     }
161.
```

SOLUTION

```
162.     public IntNode commonElements(IntNode frontL1, IntNode frontL2) {
163.         IntNode first=null, last=null;
164.         while (frontL1 != null && frontL2 != null) {
165.             if (frontL1.data < frontL2.data) {
166.                 frontL1 = frontL1.next
167.             } else if (frontL1.data > frontL2.data) {
168.                 frontL2 = frontL2.next;
169.             } else {
170.                 IntNode ptr = new IntNode(frontL1.data, null);
171.                 if (last != null) {
172.                     last.next = ptr;
173.                 } else {
174.                     first = ptr;
175.                 }
176.                 last = ptr;
```

```
177.                 frontL1 = frontL1.next;
178.                 frontL2 = frontL2.next;
179.             }
180.         }
181.         return first;
182.     }
183.
```