CSE143 Lecture #15 Problems

For these problems, assume that we are using the standard ListNode class:

```
public class ListNode {
    public int data;       // data stored in this node
    public ListNode next;  // link to next node in the list

    <constructors>
}
```

And that we are writing methods for a class called LinkedIntList that has a single data field of type ListNode called front:

```
public class LinkedIntList {
    private ListNode front;

    <methods>
}
```

In solving these problems, you may not call any other methods of the class, you may not construct new nodes and you may not use any auxiliary data structure to solve this problem (no array, ArrayList, stack, queue, String, etc). You also may not change any data fields of the nodes. You MUST solve these problems by rearranging the links of the lists involved.

1. Write a method reverse that reverses the order of the elements in the list. For example, if the list initially stores this sequence of integers:

   [1, 8, 19, 4, 17]

   It should store the following sequence of integers after reverse is called:

   [17, 4, 19, 8, 1]

2. Write a method switchPairs that switches the order of elements in a linked list of integers in a pairwise fashion. Your method should switch the order of the first two values, then switch the order of the next two, switch the order of the next two, and so on. For example, if the list initially stores these values:

   [3, 7, 4, 9, 8, 12]

   your method should switch the first pair (3, 7), the second pair (4, 9) and the third pair (8, 12), to yield this list:

   [7, 3, 9, 4, 12, 8]

   If there are an odd number of values in the list, the final element is not moved. For example, if the original list had been:

   [3, 7, 4, 9, 8, 12, 2]

   It would again switch pairs of values, but the final value (2) would not be moved, yielding this list:

   [7, 3, 9, 4, 12, 8, 2]

3. Write a method removeDuplicates that removes any duplicates from a list of integers. The resulting list should have the values in the same relative order as their first occurrence in the original list. In other words, a value i should appear before a value j in the final list if and only if the first occurrence of i appears before the first occurrence of j in the original list. For example, if a variable called list stores the following:

```
[14, 8, 14, 12, 1, 14, 11, 8, 8, 10, 4, 9, 1, 2, 5, 2, 4, 12, 12]
```

and the following call is made:

```
list.removeDuplicates();
```

then list should store these values after the call:

```
[14, 8, 12, 1, 11, 10, 4, 9, 2, 5]
```

4. Write a method takeSmallerFrom that compares two lists of integers, making sure that the first list has smaller values in corresponding positions. For example, suppose the variables list1 and list2 refer to lists that contain the following values:

```
list1: [3, 16, 7, 23]
list2: [2, 12, 6, 54]
```

If the following call is made:

```
list1.takeSmallerFrom(list2);
```

the method will compare values in corresponding positions and move the smaller values to list1. It will find that among the first pair, 2 is smaller than 3, so it needs to move. In the second pair, 12 is smaller than 16, so it needs to move. In the third pair, 6 is smaller than 7, so it needs to move. In the fourth pair, 54 is not smaller than 23, so those values can stay where they are. Thus, after the call, the lists should store these values:

```
list1: [2, 12, 6, 23]
list2: [3, 16, 7, 54]
```

One list might be longer than the other, in which case those values should stay at the end of their list. For example, for these lists:

```
list1: [2, 4, 6, 8, 10, 12]
list2: [1, 3, 6, 9]
```

the call:

```
list1.takeSmallerFrom(list2);
```

should leave the lists with these values:

```
list1: [1, 3, 6, 8, 10, 12]
list2: [2, 4, 6, 9]
```

5. Write a method called surroundWith that takes an integer x and an integer y as parameters and surrounds all nodes in the list containing the value x with new nodes containing the value y. In particular, each node that contains the value x as data should have a new node just before it that contains the value y, and each node that contains the value x as data should have a new node just after it that contains the value y. If no nodes in the list contain the value x, then the list should not be modified. For example, suppose that the variables list1, list2, and list3 store the following sequences of values:

```
list1: [0, 1, 2, 1]
list2: [0, 1, 0]
list3: [0, 1, 2]
```

and we make the following calls:

```
list1.surroundWith(1, 4);   // surround 1s with 4s
list2.surroundWith(1, 1);   // surround 1s with 1s
list3.surroundWith(3, 4);   // surround 3s with 4s
```

then list1, list2, and list3 should store the following values:

```
list1: [0, 4, 1, 4, 2, 4, 1, 4]
list2: [0, 1, 1, 1, 0]
list3: [0, 1, 2]
```

6. Write a method removeEvens that removes the values in even-numbered positions from a list, returning those values in their original order as a new list. For example, if a variable called list1 stores these values:

```
list1: [8, 13, 17, 4, 9, 12, 98, 41, 7, 23, 0, 92]
```

Then the following call:

```
LinkedIntList list2 = list1.removeEvens();
```

Should result in list1 and list2 storing the following values:

```
list1: [13, 4, 12, 41, 23, 92]
list2: [8, 17, 9, 98, 7, 0]
```

Notice that the values stored in list2 are the values that were originally in even-valued positions (index 0, index 2, index 4, and so on) and that these values appear in the same order as in the original list. Also notice that the values left in list1 also appear in the same order as in the original list.

Recall that LinkedIntList has a zero-argument constructor that returns an empty list. You may not call any methods of the class other than the constructor to solve this problem. You are not allowed to create any new nodes or to change the values stored in data fields to solve this problem. You MUST solve it by rearranging the links of the list.

CSE143 Lecture #15 Solutions

1. One possible solution appears below.

```
public void reverse() {
    ListNode current = front;
    ListNode previous = null;
    while (current != null) {
        ListNode nextNode = current.next;
        current.next = previous;
        previous = current;
        current = nextNode;
    }
    front = previous;
}
```

2. Two possible solutions appear below. The second is shorter because it is written recursively.

```
public void switchPairs() {
    if (front != null && front.next != null) {
        ListNode current = front.next;
        front.next = current.next;
        current.next = front;
        front = current;
        current = current.next;
        while (current.next != null && current.next.next != null) {
            ListNode temp = current.next.next;
            current.next.next = temp.next;
            temp.next = current.next;
            current.next = temp;
            current = temp.next;
        }
    }
}
```

```
public void switchPairs() {
    front = switchPairs(front);
}

private ListNode switchPairs(ListNode list) {
    if (list != null && list.next != null) {
        ListNode temp = list.next;
        list.next = temp.next;
        temp.next = list;
        list = temp;
        list.next.next = switchPairs(list.next.next);
    }
    return list;
}
```

3. One possible solution appears below.

```
public void removeDuplicates() {
    ListNode current = front;
    while (current != null) {
        ListNode current2 = current;
        while (current2.next != null) {
            if (current2.next.data == current.data) {
                current2.next = current2.next.next;
            } else {
                current2 = current2.next;
            }
        }
        current = current.next;
    }
}
```

4. One possible solution appears below.

```
public void takeSmallerFrom(LinkedIntList other) {
    if (front != null && other.front != null) {
        if (front.data > other.front.data) {
            ListNode temp = front;
            front = other.front;
            other.front = temp;
            temp = front.next;
            front.next = other.front.next;
            other.front.next = temp;
        }
        ListNode current1 = front;
        ListNode current2 = other.front;
        while (current1.next != null && current2.next != null) {
            if (current1.next.data > current2.next.data) {
                ListNode temp = current1.next;
                current1.next = current2.next;
                current2.next = temp;
                temp = current1.next.next;
                current1.next.next = current2.next.next;
                current2.next.next = temp;
            }
            current1 = current1.next;
            current2 = current2.next;
        }
    }
}
```

5. One possible solution appears below.

```
public void surroundWith(int x, int y) {
    if (front != null) {
        ListNode current = front;
        if (front.data == x) {
           front.next = new ListNode(y, front.next);
           front = new ListNode(y, front);
           current = front.next.next;
        }
        while (current.next != null) {
           if (current.next.data == x) {
              current.next.next = new ListNode(y, current.next.next);
              current.next = new ListNode(y, current.next);
              current = current.next.next.next;
           } else
              current = current.next;
        }
     }
}
```

6. One possible solution appears below.

```
    public LinkedIntList removeEvens() {
        LinkedIntList result = new LinkedIntList();
        if (front != null) {
            result.front = front;
            front = front.next;
            ListNode current = front;
            ListNode resultLast = result.front;
            while (current != null && current.next != null) {
                resultLast.next = current.next;
                resultLast = current.next;
                current.next = current.next.next;
                current = current.next;
            }
            resultLast.next = null;
        }
        return result;
    }
```