RUTGERS

CS112 Data Structures

Recitation 02

Yu Yang
yy388@cs.rutgers.edu
Office: CoRE 331
Office Hour: 3-5pm, Wed.

Outline

- Problem Set 2
- Debug

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

```
public class IntNode {
   public int data;
   public IntNode next;
   public IntNode(int data, IntNode next) {
       this.data = data; this.next = next;
   }
   public String toString() {
       return data + "";
}
```

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:

```
public static IntNode addBefore(IntNode front, int target, int newItem) {
      /* COMPLETE THIS METHOD */
}
```

SOLUTION

```
        public static IntNode addBefore(IntNode front, int target, int newItem) {
            IntNode prev=null, ptr=front;
            while (ptr != null && ptr.data != target) {
                prev = ptr;
  ptr = ptr.next;
  }
            if (ptr == null) { // target not found
                return front;
            }
            IntNode temp = Intnew Node(newItem, ptr); // next of new node should point to target
            if (prev == null) { // target is first item, so new node will be new front
               return temp;
            }
            prev.next = temp;
            return front;  // front is unchanged
        }
```

5. * 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:

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

SOLUTION

```
public static StringNode deleteAllOcurrences(StringNode front, String target) {

  if (front == null) {
     return null;
  }

  StringNode curr=front, prev=null;

  while (curr != null) {
     if (curr.data.equals(target)) {
        if (prev == null) {        // target is the first element
           front = curr.next;
        } else {
           prev.next = curr.next;
        }
     } else {
        prev = curr;
     }
     curr = curr.next;
  }

  return front;
}
```

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

should produce a new linked list:

```
3->12
```

You may assume that the original lists do not have any duplicate items.

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

```
public class IntNode {
   public int data;
   public IntNode next;
   public IntNode(int data, IntNode next) {
       this.data = data; this.next = next;
   }
   public String toString() {
       return data + "";
}
```

Complete the following method:

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

SOLUTION

```
public IntNode commonElements(IntNode frontL1, IntNode frontL2) {
   IntNode first=null, last=null;
   while (frontL1 != null && frontL2 != null) {
      if (frontL1.data < frontL2.data) {
         frontL1 = frontL1.next
      } else if (frontL1.data > frontL2.data) {
         frontL2 = frontL2.next;
      } else {
         IntNode ptr = new IntNode(frontL1.data, null);
         if (last != null) {
            last.next = ptr;
         } else {
            first = ptr;
         }
         last = ptr;
         frontL1 = frontL1.next;
         frontL2 = frontL2.next;
      }
   }
   return first;
}
```