Com S 228
Spring 2018
Exam 2

DO NOT OPEN THIS EXAM UNTIL INSTRUCTED TO DO SO

Name: ____

ISU NetID (username): _______________________________________@iastate.edu

Recitation section **(please circle one):**

1.	R	10:00 am	(Xinxin, Tammay)
2.	R	2:10 pm	(Mike P., Arnoldo)
3.	R	1:10 pm	(Gabriel, Nirala)
4.	R	4:10 pm	(Christine, Nirala)
5.	R	3:10 pm	(Jason, Xiaoqian)
6.	T	9:00 am	(Jacob, Waqwoya)
7.	T	2:10 pm	(Mike L., Jason)
8.	T	10:00am	(Andrew, Waqwoya)

Closed book/notes, no electronic devices, no headphones. Time limit **60 minutes**.

Partial credit may be given for partially correct solutions.

- Use correct Java syntax for writing code.
- You are not required to write comments for your code; however, brief comments may help make your intention clear in case your code is incorrect.

If you have questions, please ask!

Question	Points	Your Score
1	28	
2	16	
3	18	
4	38	
Total	100	

1. (28 pts) The `main()` method below executes a code snippet after the initialization of a `List` object. On the next page, you will see several snippets of code, each to be executed ***within a separate call*** of the `main()` method.

```
public static void main(String[] args) throws NoSuchElementException,
                                              IllegalStateException
{
      List<String> aList;
      ListIterator<String> iter, iter2;

      // initialization
      aList = new ArrayList<String>();
      aList.add("A");
      aList.add("B");
      aList.add("X");
      aList.add("C");
      aList.add("D");

      // code snippet
      // ...
}
```

Note that each snippet is *separate* and executed *independently* right after the initialization.

For each snippet,

a) show what the ***output*** from the `println` statement is, if any, and

b) draw the ***state*** of `aList` and the ***iterator*** after the code executes, and

c) do ***not*** display any other list that may appear in the code.

However, if the code throws an exception, do ***not*** draw the list but instead write down the exception that is thrown. In this case, ***also show the output***, if any.

Use a bar (**|**) symbol to indicate the iterator's logical cursor position. For example, right after the statement

```
      iter = aList.listIterator();
```

the list would be drawn as follows.

```
|A B X C D
```

(the first one has been done for you as an example). If two iterators appear in the code, draw the second iterator as a dashed bar like ¦ or an upward arrow like ↑.

Suggestion: For ***partial credit***, you may also want to draw the intermediate states of the list and iterator after executing every one or few lines of code in a snippet.

Code snippet	**Output**	**List and iterator state, or exception thrown**
`iter = aList.listIterator();`	(none)	|A B X C D

`// 3 pts`
`iter = aList.listIterator(2);`
`System.out.println(iter.previous());`		

`// 3 pts`
`iter = aList.listIterator(aList.size());`
`iter.remove();`		

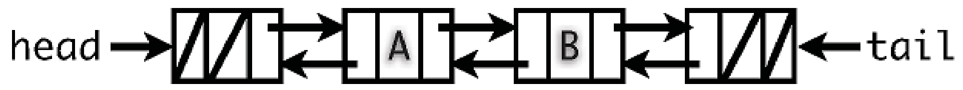
`// 4 pts`
`iter = aList.listIterator();`
`while` `(iter.hasNext())`
`{`
`    iter.set(iter.next() + iter.previous());`
`    System.out.println(iter.next());`
`}`		

`// 5 pts`
`iter = aList.listIterator();`
`while` `(iter.hasNext())`
`{`
`    iter.add(iter.next());`
`    System.out.println(iter.previous());`
`    iter.next();`
`}`		

`// 6 pts`
`iter = aList.listIterator();`
`iter2 = aList.listIterator(aList.size());`
`while` `(iter.nextIndex() < iter2.previousIndex())`
`{`
`    String s = iter.next();`
`    String t = iter2.previous();`
`    iter.set(t);`
`    iter2.set(s);`
`}`		

`// 7 pts`
`iter = aList.listIterator();`
`iter2 = aList.listIterator(1);`
`while` `(iter2.hasNext())`
`{`
`    iter.next();`
`    iter.set(iter2.next());`
`    System.out.println(iter.previous());`
`}`		

2. (16 pts) For the next questions assume a ***doubly-linked list*** implementation of the List interface that includes a head node and a tail node. Assume that this includes an implementation of all methods of the `ListIterator` interface. The list has n elements.



a) (5 pts) Give the big-*O* time complexity of the following **List** API operation.

```
public boolean remove(Object obj)
```

b) (5 pts) Give the big-O time complexity of the following **ListIterator** API operation.

```
public void set(E item)
```

c) (6 pts) Suppose that an iterator is created at a given position. Then a total of k calls to the **ListIterator** API methods `add()`, `remove()`, or `set()` are performed. Between every two adjacent such calls are $O(1)$ **ListIterator** calls to `next()` or `previous()`. What is the big-O time complexity of the iterator creation and the *k* calls together? Note that either $k \leq n$ or $k > n$ is possible.

3. (18 pts) Infix and postfix expressions.

a) (9 pts) Convert the following infix expression into postfix by filling the row of boxes below ***from left to right***, with each filled box containing ***exactly one*** operand, operator, or parenthesis. (You may end up with some unfilled boxes on the right).

```
(a * b + 5) ^ (c - d % (e + 2 ^ f) / g) ^ 3 - (h + i)
```

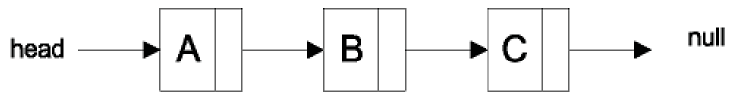
Postfix:

b) (9 pts) Convert the following postfix expression into infix by filling the row of boxes below in the same way as specified for part a). (Parentheses may be needed, and unfilled boxes are possible.)

```
1 a 2 b 3 c 4 d 5 e / + / + / + / + /
```

Infix:

4. (38 pts) Recall the class `SinglyLinkedCollection<E>` that implements the `Collection` interface based on a null terminated, singly-linked list with no dummy node. A list example with three nodes is shown below. Also shown is part of the class implementation that is relevant to this problem.



```
public class SinglyLinkedCollection<E> extends AbstractCollection<E>

{
      private Node head;
      private int size;

      // …

      private class Node

      {
            public E data;
            public Node next;

            public Node(E pData, Node pNext)
            {
                  data = pData;
                  next = pNext;
            }
      }

      // …
}
```

Within the class, add a method `mergeLists(list1, list2, comp)`, where `comp` is a supplied `Comparator` object defined in the class `E` or some superclass of `E`. The method merges two ordered lists `list1` and `list2` into a new ordered list, without modifying `list1` and `list2`. (Picture yourself implementing the merge step in mergesort, which now operates on linked lists rather than arrays.)

The template for the method is given on the next pages. Pay attention to the following instructions.

- In an ordered list, items are in ***non-decreasing order*** according to the supplied comparator.
- There is ***no need*** to verify that `list1` and `list2` are ordered.
- ***Fill a wildcard type*** in the blank preceding the third parameter `comp`.
- *Note that one or both of the input lists may be **empty**.*
- For your convenience, the implementation breaks down into ***four steps***.
- The comments before each step outline what the step does. It is helpful to follow them.

- ***Do not use iterators***.
- It may help to draw a picture to work out the link updates.

```
public SinglyLinkedCollection<E>
mergeLists(SinglyLinkedCollection<E> list1, SinglyLinkedCollection<E> list2,

      ________________________________________________ comp)  // (4 pts)

{
      // The sorted list to combine items from list1 and list2.
      SinglyLinkedCollection<E> list3 = new SinglyLinkedCollection<E>();

      // The next two variables will be used to scan the two lists.
      Node cur1 = list1.head;
      Node cur2 = list2.head;


      // 1. Initialization (10 pts)
      //
      // Let list3.head reference the smallest item from the two lists.
      // Note that one or both lists may be empty.
      // 2. Merging (10 pts)
      //
      // Iterate the two references cur1 and cur2 through the remaining nodes.
      // Construct list3 on the fly by creating new nodes using the reference cur3.

      Node cur3 = list3.head; // This variable is used for generating nodes of list3.


      // Both lists have unprocessed elements.
```

```
// 3. List appending (10 pts)
//
// Step 2 stops when the end of one list is reached (thus all of its items have
// been added to list3 at this point).  Append the remainder of the other list
// to list3.



// All the items from list1 have been added to list3.
```

```
}






        // All the items from list2 have been added to list3.






        // 4. Updates of instance variables, if any, of the merged list. (4 pts)





    }
```