CSE 143, Winter 2012
Midterm Exam
Monday February 13, 2012

Personal Information:

Name: ______________________________

Section: ____________________ **TA:** ____________________

Student ID #: ____________________

- You have 50 minutes to complete this exam.
  You may receive a deduction if you keep working after the instructor calls for papers.
- This exam is open-book for our *Building Java Programs* textbook but closed to any other papers, books, notes, or other documents. You may not use any computing devices including calculators.
- Code will be graded on proper behavior/output and not on style, unless otherwise indicated.
- Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks.

  The only abbreviations that *are* allowed for this exam are:
  - `S.o.p` for `System.out.print`, and
  - `S.o.pln` for `System.out.println`.
- You do not need to write `import` statements in your code.
- If you enter the room, you must turn in an exam before leaving the room.
- You must show your Student ID to a TA or instructor for your exam to be accepted.

Good luck!

Score summary: (for grader only)

Problem	Description	Earned	Max
1	ArrayList Mystery		15
2	Recursive Tracing		20
3	Collections		20
4	Stacks and Queues		15
5	Linked Lists		15
6	Recursive Programming		15
TOTAL	**Total Points**		**100**

1. `ArrayList` Mystery

Consider the following method:

```
public static void mystery(ArrayList<Integer> list) {
    for (int index = 0; index < list.size(); index++) {
        int elementValue = list.remove(index);
        if (elementValue % 2 == 0) {
            list.add(index);
        }
    }
    System.out.println(list);
}
```

Write the output produced by the method when passed each of the following `ArrayList`s:

List	Output
a) `[5, 2, 5, 2]`	
b) `[3, 5, 8, 9, 2]`	
c) `[0, 1, 4, 3, 1, 3]`	

2. Recursive Tracing

For each of the calls to the following recursive method below, indicate what output is produced:

```
public static void mystery(int x, int y) {
    if (y <= 0) {
        System.out.print("0 ");
    } else if (x > y) {
        System.out.print(x + " ");
        mystery(x - y, y);
    } else {
        mystery(x, y - x);
        System.out.print(y + " ");
    }
}
```

Call	Output
a) `mystery(6, 3);`	
b) `mystery(2, 3);`	
c) `mystery(5, 8);`	
d) `mystery(21, 12);`	
e) `mystery(3, 10);`	

3. Collections

Write a method **`byAge`** that accepts three parameters: 1) a `Map` where each key is a person's name (a string) and the associated value is that person's age (an integer); 2) an integer for a minimum age; and 3) an integer for a max age. Your method should return a new map with information about people with ages between the min and max, inclusive.

In your result map, each key is an integer age, and the value for that key is a string with the names of all people at that age, separated by `"and"` if there is more than one person of that age. Include only ages between the min and max inclusive, where there is at least one person of that age in the original map. If the map passed in is empty, or if there are no people in the map between the min/max ages, return an empty map.

For example, if a `Map` named `ages` stores the following key=value pairs:

```
{Paul=28, David=20, Janette=18, Marty=35, Stuart=98, Jessica=35, Helene=40,
 Allison=18, Sara=15, Grace=25, Zack=20, Galen=15, Erik=20, Tyler=6, Benson=48}
```

The call of `byAge(ages, 16, 25)` should return the following map (the contents can be in any order):

```
{18=Janette and Allison, 20=David and Zack and Erik, 25=Grace}
```

For the same map, the call of `byAge(ages, 20, 40)` should return the following map:

```
{20=David and Zack and Erik, 25=Grace, 28=Paul, 35=Marty and Jessica, 40=Helene}
```

For full credit, obey the following restrictions in your solution. A solution that disobeys them can get partial credit.

- You will need to construct a map to store your results, but you may not use any other structures (arrays, lists, etc.) as auxiliary storage. (You can have as many simple variables as you like.)
- You should not modify the contents of the map passed to your method.
- Your solution should run in no worse than O(N log N) time, where N is the number of pairs in the map.

4. Stacks and Queues

Write a method **`isSorted`** that accepts a stack of integers as a parameter and returns `true` if the elements in the stack occur in ascending (non-decreasing) order from top to bottom, and `false` otherwise. That is, the smallest element should be on top, growing larger toward the bottom. For example, passing the following stack should return `true`:

```
bottom [20, 20, 17, 11, 8, 8, 3, 2] top
```

The following stack is *not* sorted (the 15 is out of place), so passing it to your method should return a result of `false`:

```
bottom [18, 12, 15, 6, 1] top
```

An empty or one-element stack is considered to be sorted. When your method returns, the stack should be in the same state as when it was passed in. In other words, if your method modifies the stack, you must restore it before returning.

For full credit, obey the following restrictions in your solution. A solution that disobeys them can get partial credit.

- You may use **one queue or one stack** (but not both) as auxiliary storage.
  You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
- Use the `Queue` interface and `Stack`/`LinkedList` classes discussed in class.
- Use stacks/queues in stack/queue-like ways only. Do not use index-based methods such as `get`, `search`, or `set`, or for-each loops or iterators. You may call `add`, `remove`, `push`, `pop`, `peek`, `isEmpty`, and `size`.
- Your solution should run in O(*N*) time, where *N* is the number of elements of the stack.

You have access to the following two methods and may call them as needed to help you solve the problem:

```
public static void s2q(Stack<Integer> s, Queue<Integer> q) { ... }
public static void q2s(Queue<Integer> q, Stack<Integer> s) { ... }
```

5. Linked Lists

Write a method **`minToFront`** to be added to the `LinkedIntList` class. Your method should move the smallest element value to the front of the linked list. Suppose a `LinkedIntList` variable named `list` stores these elements:

```
[7, 9, 12, 5, 3, 17]
```

If you made the call of `list.minToFront();` , the list would then store the elements:

```
[3, 7, 9, 12, 5, 17]
```

If the list has more than one occurrence of the minimum value, the first occurrence is the only one that should be moved. For example, `[7, 9, 3, 12, 5, 3, 17, 3]` becomes `[3, 7, 9, 12, 5, 3, 17, 3]`. If the list is empty, calling your method should have no effect.

For full credit, obey the following restrictions in your solution. A solution that disobeys them can get partial credit.

- Do not call any other methods on the `LinkedIntList` object, such as `add`, `remove`, or `size`.
- Do not create new `ListNode` objects (though you may have as many `ListNode` variables as you like).
- Do not use other data structures such as arrays, lists, queues, etc.
- Your solution should run in O(*N*) time, where *N* is the number of elements of the linked list.

Assume that you are adding this method to the `LinkedIntList` class (that uses the `ListNode` class) below.

```
public class LinkedIntList {
    private ListNode front;
    ...
```

```
public class ListNode {
    public int data;
    public ListNode next;
    ...
```

6. Recursive Programming

Write a recursive method **`moveToEnd`** that accepts a `String` *s* and a `char` *c* as parameters, and returns a new `String` similar to *s* but with all occurrences of *c* moved to the end. The relative order of the other characters should be unchanged from their order in the original string *s*. If *s* does not contain *c*, it should be returned unmodified.

The following table shows calls to your method and their return values. Occurrences of *c* are underlined for clarity.

Call	Returns
`moveToEnd("hello", 'l')`	`"heoll"`
`moveToEnd("hello", 'e')`	`"hlloe"`
`moveToEnd("hello there", 'e')`	`"hllo threee"`
`moveToEnd("hello there", 'q')`	`"hello there"`
`moveToEnd("HELLO there", 'e')`	`"HELLO three"`
`moveToEnd("", 'x')`	`""`

You may not construct any structured objects other than `String`s (no array, `List`, `Scanner`, etc.). You will examine the `String` one character at a time but you may not use any loops to solve this problem; you must use recursion.

(extra writing space)

(extra writing space)