CSE 143 Sample Midterm Exam #6 (10wi)

1. `ArrayList` Mystery

Consider the following method:

```
public static void mystery(ArrayList<Integer> list) {
    for (int i = 1; i < list.size(); i += 2) {
        if (list.get(i - 1) >= list.get(i)) {
            list.remove(i);
            list.add(0, 0);
        }
    }
    System.out.println(list);
}
```

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

List	Output
a) `[10, 20, 10, 5]`	
b) `[8, 2, 9, 7, -1, 55]`	
c) `[0, 16, 9, 1, 64, 25, 25, 14, 0]`	

2. Recursive Tracing

For each of the calls to the following recursive method below, indicate what value is returned:

```
public static int mystery(int n, int k) {
    if (k == 0 || k == n) {
        return 1;
    } else if (k > n) {
        return 0;
    } else {
        return mystery(n - 1, k - 1) + mystery(n - 1, k);
    }
}
```

Call	Returns
a) `mystery(7, 1)`	
b) `mystery(4, 2)`	
c) `mystery(4, 3)`	
d) `mystery(5, 3)`	
e) `mystery(5, 4)`	

3. `Comparable`

Suppose you have a pre-existing class `Dieter` that models a person trying to monitor diet and weight to improve his/her Body Mass Index (BMI) rating. The class has the following data and behavior:

Field/Constructor/Method	Description
`private String name`	the dieter's full name, such as `"John Smith"`
`private int height`	the dieter's height, in inches
`private int weight`	the dieter's weight, in pounds
`public Dieter(String name, int h, int w)`	makes a dieter with given name, height, weight
`public String getName()`	returns the dieter's name
`public int getHeight()`	returns the dieter's height
`public int getWeight()`	returns the dieter's weight
`public double getBMI()`	computes the dieter's Body Mass Index (BMI), which is $(weight / height^2) \cdot 703$
`public void gain(int pounds)`	called when the dieter gains weight
`public void lose(int pounds)`	called when the dieter loses weight

Make `Dieter` objects comparable to each other using the `Comparable` interface. Add any necessary code below, and/or make any changes to the existing code headings shown.

Dieters are primarily compared by their **BMI ratings**; a dieter with a lower BMI is "less than" one with a higher BMI. If two dieters have the same BMI, the tie is broken by **height**; the shorter person is considered to be "less than" the taller one. If two dieters have the same BMI and height, the tie is broken by **name**. The dieter whose name comes first in alphabetical order is considered "less than" the one whose name comes later in ABC order. If the two dieters have the same BMI, height, and name, they are considered to be "equal." Your method should not modify any dieter's state. You may assume the parameter passed is not `null`.

```
public class Dieter {
    ...
    // write any added code here









}
```

4. Stacks and Queues

Write a method **`expunge`** that accepts a stack of integers as a parameter and makes sure that the stack's elements are in ascending order from top to bottom, by removing from the stack any element that is smaller than any element(s) on top of it. For example, suppose a variable `s` stores the following elements:

```
bottom [4, 20, 15, 15, 8, 5, 7, 12, 3, 10, 5, 0] top
```

The element values `3`, `7`, `5`, `8`, and `4` should be removed because each has an element above it with a larger value. So the call of `expunge(s);` should change the stack to store the following elements in this order:

```
bottom [20, 15, 15, 12, 10, 5, 0] top
```

Notice that now the elements are in non-decreasing order from top to bottom. If the stack is empty or has just one element, nothing changes. You may assume that the stack passed is not `null`.

(*Hint:* An element *e* that should be removed is one that is smaller than some element above *e*. But since the elements above *e* are in sorted order, you may not need to examine all elements above *e* in order to know whether to remove *e*.)

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 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 lecture.
- Use stacks/queues in stack/queue-like ways only. Do not call index-based methods such as `get`, `search`, or `set` (or for-each) on a stack/queue. You may call only `add`, `remove`, `push`, `pop`, `peek`, `isEmpty`, and `size`.

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. Collections

Write a method **`commonFirstLetters`** that accepts a list of strings as a parameter, and returns a `Set` of strings. Your method should examine the strings in the list passed and return a set of all first letters that occur more than once. In other words, if two or more strings in the list begin with a particular character (case-insensitively), that letter should be part of the set that you return.

Consider a list variable called `list` containing the following elements:

```
["hi", "how", "is", "He", "Marty!", "this", "morning?", "fine.", "?huh?", "HOW", "I"]
```

One word in the list begins with `"f"`, four begin with `"h"`, two begin with `"i"`, two begin with `"m"`, one begins with `"t"`, and one begins with `"?"`. Therefore the set you return should contain the following elements:

```
["h", "i", "m"]
```

If no first character occurs 2 or more times, return an empty set. The order the elements appear in the set does not matter. You may assume that the list passed is not `null` and that none of its elements are `null` or empty strings.

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

- You may create only **up to two (2) new data structures of your choice** (list, stack, queue, set, map, etc.) as auxiliary storage. (You can have as many simple variables as you like.)
- Your solution should run in less than $O(N^2)$ time, where N is the number of strings in the list.
- You should not modify the contents of the list passed to your method.

---

6. Linked Lists

Write a method **`removeMin`** that could be added to the `LinkedIntList` class that removes the smallest element value from the linked list. Suppose a `LinkedIntList` variable named `list` stores the following elements:

```
[8, 4, 7, 2, 9, 4, 5, 3]
```

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

```
[8, 4, 7, 9, 4, 5, 3]
```

If the list has more than one occurrence of the minimum value, the first occurrence is the one that should be removed. For example, `[8, 10, 13, 8, 8, 14, 9, 8, 11]` becomes `[10, 13, 8, 8, 14, 9, 8, 11]`. If the list is empty, you should throw a `NoSuchElementException`.

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;
    ...
```