CSE143 Midterm
Summer 2017

Name of Student: _____

Section (e.g., AA):_____ Student Number: _____

The exam is divided into six questions with the following points:

#	Problem Area	Points	Score
1	Recursive Tracing	15	_____
2	Recursive Programming	15	_____
3	ListNodes	15	_____
4	Collections	20	_____
5	Stacks/Queues	25	_____
6	ArrayIntList Programming	10	_____
	Total	100	_____

This is a closed-book/closed-note exam. Space is provided for your answers. There is a "cheat sheet" at the end that you can use as scratch paper. You are not allowed to access any of your own papers during the exam.
You may not use calculators or any other devices.

The exam is not, in general, graded on style and you do not need to include comments. For the stack/queue and collections questions, however, you are expected to use generics properly and to declare variables using interfaces when possible. You may only use the Stack and Queue methods on the cheat sheet, which are the methods we discussed in class. You are not allowed to use programming constructs like break, continue, or returning from a void method on this exam.

Do not abbreviate code, such as "ditto" marks or dot-dot-dot ... marks.
The only abbreviations you are allowed to use for this exam are:

```
S.o.p for System.out.print
S.o.pln for System.out.println
```

You are NOT to use any electronic devices while taking the test, including calculators. Anyone caught using an electronic device will receive a 10 point penalty.

Do not begin work on this exam until instructed to do so. Any student who starts early or who continues to work after time is called will receive a 10 point penalty.

If you finish the exam early, please hand your exam to the instructor and exit quietly through the front door.

1. **Recursive Tracing, 15 points:** Consider the following method:

```
public int mystery(int n, int m) {
    if (n == 0 || m == 0) {
        return 0;
    } else if (n % 10 == m % 10) {
        return 1 + mystery(n / 10, m / 10);
    } else {
        return mystery(n / 10, m / 10);
    }
}
```

For each call below, indicate what output is produced:

Method Call	**Value Returned**
mystery(18, 0)	_____
mystery(8, 18)	_____
mystery(25, 21)	_____
mystery(305, 315)	_____
mystery(20734, 1724)	_____

2. **Recursive Programming, 15 points:** Write a recursive method called writeNumbers that takes an integer n as a parameter and that writes the first n numbers separated by commas with the odd numbers in descending order followed by the even numbers in ascending order. For example, the call:

```
writeNumbers(5);
```

should produce the following output:

```
5, 3, 1, 2, 4
```

The odd numbers (5, 3, and 1) appear first in descending order followed by the even numbers (2 and 4) in ascending order. Notice that commas are used to separate consecutive values in the list. Your method should send its output to System.out and should not call println to complete the line of output.

For example, the following calls:

```
writeNumbers(5);
System.out.println();  // to complete the line of output
writeNumbers(1);
System.out.println();  // to complete the line of output
writeNumbers(8);
System.out.println();  // to complete the line of output
```

should produce exactly three lines of output:

```
5, 3, 1, 2, 4
1
7, 5, 3, 1, 2, 4, 6, 8
```

Your method should throw an IllegalArgumentException if passed a value less than 1. You are not allowed to construct any structured objects (no array, ArrayList, String, StringBuilder, etc) and you may not use a while loop, for loop, or do/while loop to solve this problem; you must use recursion.

3. **Linked Lists, 15 points**: Fill in the "code" column in the following table providing a solution that will turn the "before" picture into the "after" picture by modifying links between the nodes shown. You are not allowed to change any existing node's data field value and you are not allowed to construct any new nodes, but you are allowed to declare and use variables of type ListNode (often called "temp" variables). You are limited to at most two variables of type ListNode for each of the four subproblems below.

You are writing code for the ListNode class discussed in lecture:

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

As in the lecture examples, all lists are terminated by null and the variables p and q have the value null when they do not point to anything.

before	after	code
p->[1]->[2] q->[3]	p->[2] q->[3]->[1]	
p->[1] q->[2]->[3]	p->[3] q->[1]->[2]	
p->[1]->[2] q->[3]->[4]	p->[3]->[2]->[1] q->[4]	
p->[1]->[2] q->[3]->[4]->[5]	p->[1]->[3]->[5] q->[2]->[4]	

4. **Collections Programming, 20 points:** Write a method called takingAlongside that takes a class name and an enrollment map as parameters and that returns a set that contains the names of all classes taken by the students in the given class. The enrollment map uses student names as keys (strings) and has sets of class names as values (also strings).

   For example, a variable called enrollments might contain the following map:

   ```
   {"Ian"=["CSE 143", "CSE 331"], "Jin"=["CSE 143", "GH 101", "MATH 308"],
    "Aaron"=["CSE 331", "GH 101"], "Kyle"=["CSE 331", "PHYS 121"],
    "Miri"=["CSE 143"], "Zach"=["CSE 446", "PHIL 100"], "Ayaz"=["CSE 142"]}
   ```

   This map indicates, for example, that Ian is taking CSE 143 and CSE 331 and that Ayaz is taking CSE 142. Suppose that the following call is made:

   ```
   takingAlongside("CSE 143", enrollments)
   ```

   Given this call, the method would return a set containing all the classes taken by students who are taking CSE 143. In the example above, Ian, Jin, and Miri are taking CSE 143. Therefore, the method should return a set containing all of the classes those students are taking, including CSE 143 itself. Thus, it would return:

   ```
   ["CSE 143", "CSE 331", "GH 101", "MATH 308"]
   ```

   Another way of thinking about what this method is doing is that each time it encounters a student taking the target class, it includes all of the classes that student is taking in the overall answer.

   The set returned by the method should be ordered alphabetically. If no student is taking the given class, the method should return the empty set.

   You may assume that the given string and map are not null and none of the map's keys or values are null or reference null elements. The method should not modify the provided map or any of the structures it references.

   You can use space on the next page to write your answer.

This page is left blank so you have extra space on #4

5. **Stacks/Queues, 25 points:** Write a method called compressDuplicates that takes a stack of integers as a parameter and that replaces each sequence of duplicates with a pair of values representing a count of the number of duplicates followed by the number. For example, suppose a variable called s stores the following sequence of values:

```
bottom [2, 2, 2, 2, 2, -5, -5, 3, 3, 3, 3, 4, 4, 1, 0, 17, 17] top
```

and we make the following call:

```
compressDuplicates(s);
```

Then s should store the following values after the call:

```
bottom [5, 2, 2, -5, 4, 3, 2, 4, 1, 1, 1, 0, 2, 17] top
```

This new stack indicates that the original had 5 occurrences of 2 at the bottom of the stack followed by 2 occurrences of -5 followed by 4 occurrences of 3, and so on. This process works best when there are many duplicates in a row. For example, if the stack instead had stored:

```
bottom [10, 20, 10, 20, 10, 20] top
```

Then the resulting stack ends up being longer than the original:

```
bottom [1, 10, 1, 20, 1, 10, 1, 20, 1, 10, 1, 20] top
```

If the stack is empty, your method should not change it. You are to use one queue as auxiliary storage to solve this problem. You may not use any other auxiliary data structures to solve this problem, although you can have as many simple variables as you like. You also may not solve the problem recursively. Your solution must run in O(n) time where n is the size of the stack. Use the Stack and Queue structures described in the cheat sheet and obey the restrictions described there.

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 s, Queue q) { ... }
public static void q2s(Queue q, Stack s) { ... }
```

You may use the space on the next page to write your answer.

This page is left blank so you have extra space on #5

6. **ArrayIntList Programming, 10 points:** Write a method called removeFront for the ArrayIntList class that takes an integer n as a parameter and that removes the first n values from a list of integers.

   For example, if a variable called list stores this sequence:

   ```
   [8, 17, 9, 24, 42, 3, 8]
   ```

   and the following call is made:

   ```
   list.removeFront(4);
   ```

   then it should store the following values after the call:

   ```
   [42, 3, 8]
   ```

   Notice that the first four values in the list have been removed and the other values appear in the same order as in the original list.

   You are writing a method for the ArrayIntList class discussed in lecture:

   ```
   public class ArrayIntList {
       private int[] elementData; // list of integers
       private int size;          // current # of elements in the list

       <methods>
   }
   ```

   Your method should throw an IllegalArgumentException if the parameter n is less than 0 or greater than the number of elements in the list. You are not to call any other ArrayIntList methods to solve this problem, you are not allowed to define any auxiliary data structures (no array, ArrayList, etc), and your solution must run in O(n) time where n is the length of the list.

^_^ CSE 143 MIDTERM EXAM CHEAT SHEET ^_^

Constructing Various Collections

```
List<Integer> list = new ArrayList<Integer>();
Queue<Double> queue = new LinkedList<Double>();
Stack<String> stack = new Stack<String>();
Set<String> words = new HashSet<String>();
Map<String, Integer> counts = new TreeMap<String, Integer>();
```

Methods Found in ALL collections (Lists, Stacks, Queues, Sets, Maps)

`equals(`**collection**`)`	returns `true` if the given other collection contains the same elements
`isEmpty()`	returns `true` if the collection has no elements
`size()`	returns the number of elements in the collection
`toString()`	returns a string representation such as `"[10, -2, 43]"`

Methods Found in both Lists and Sets (ArrayList, LinkedList, HashSet, TreeSet)

`add(`**value**`)`	adds value to collection (appends at end of list)
`addAll(`**collection**`)`	adds all the values in the given collection to this one
`contains(`**value**`)`	returns `true` if the given value is found somewhere in this collection
`iterator()`	returns an Iterator object to traverse the collection's elements
`clear()`	removes all elements of the collection
`remove(`**value**`)`	finds and removes the given value from this collection
`removeAll(`**collection**`)`	removes any elements found in the given collection from this one
`retainAll(`**collection**`)`	removes any elements *not* found in the given collection from this one

`List<E>` Methods (10.1)

`add(`**index**`, `**value**`)`	inserts given value at given index, shifting subsequent values right
`indexOf(`**value**`)`	returns first index where given value is found in list (-1 if not found)
`get(`**index**`)`	returns the value at given index
`lastIndexOf(`**value**`)`	returns last index where given value is found in list (-1 if not found)
`remove(`**index**`)`	removes/returns value at given index, shifting subsequent values left
`set(`**index**`, `**value**`)`	replaces value at given index with given value
`subList(`**from**`, `**to**`)`	returns sub-portion at indexes **from** (inclusive) and **to** (exclusive)

`Stack<E>` Methods

`peek()`	returns the top value from the stack without removing it
`pop()`	removes the top value from the stack and returns it; `peek`/`pop` throw an `EmptyStackException` if the stack is empty
`push(`**value**`)`	places the given value on top of the stack

`Queue<E>` Methods

`add(`**value**`)`	places the given value at the back of the queue
`peek()`	returns the front value from the queue without removing it; returns `null` if the queue is empty
`remove()`	removes the value from the front of the queue and returns it; throws a `NoSuchElementException` if the queue is empty

^_^ CSE 143 MIDTERM EXAM CHEAT SHEET ^_^

`Map<K, V>` Methods (11.3)

`containsKey(`**`key`**`)`	`true` if the map contains a mapping for the given key
`get(`**`key`**`)`	the value mapped to the given key (`null` if none)
`keySet()`	returns a `Set` of all keys in the map
`put(`**`key, value`**`)`	adds a mapping from the given key to the given value
`putAll(`**`map`**`)`	adds all key/value pairs from the given map to this map
`remove(`**`key`**`)`	removes any existing mapping for the given key
`toString()`	returns a string such as `"{a=90, d=60, c=70}"`
`values()`	returns a `Collection` of all values in the map

`String` Methods (3.3, 4.4)

`charAt(`**`i`**`)`	the character in this String at a given index
`contains(`**`str`**`)`	`true` if this String contains the other's characters inside it
`endsWith(`**`str`**`)`	`true` if this String ends with the other's characters
`equals(`**`str`**`)`	`true` if this String is the same as *str*
`equalsIgnoreCase(`**`str`**`)`	`true` if this String is the same as *str*, ignoring capitalization
`indexOf(`**`str`**`)`	first index in this String where given String begins (-1 if not found)
`lastIndexOf(`**`str`**`)`	last index in this String where given String begins (-1 if not found)
`length()`	number of characters in this String
`startsWith(`**`str`**`)`	`true` if this String begins with the other's characters
`substring(`**`i, j`**`)`	characters in this String from index *i* (inclusive) to *j* (exclusive)
`toLowerCase(), toUpperCase()`	a new String with all lowercase or uppercase letters