CSE143 Midterm
Autumn 2018

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.

After finishing your exam, make sure to fill out the bonus question on the next page if you are participating in the extra credit opportunity.

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 void mystery(int x) {
    if (x < 10) {
        System.out.print(x);
    } else {
        int y = x % 10;
        System.out.print(y);
        mystery(x / 10);
        System.out.print(y);
    }
}
```

For each call below, indicate what output is produced:

Method Call	**Output Produced**
mystery(7)	_____
mystery(38)	_____
mystery(194)	_____
mystery(782)	_____
mystery(3842)	_____

2. **Recursive Programming, 15 points:** Write a recursive method called sameDashes that takes two equal length strings as parameters and that returns whether or not they have dashes in the same places (returning true if they do and returning false otherwise). For example, below are four pairs of strings of equal length that have the same pattern of dashes. Notice that the last pair has no dashes at all.

string1:	"hi--there-you."	"-15-389"	"criminal-plan"	"abc"
string2:	"12--(134)-7539"	"-xy-zzy"	"(206)555-1384"	"9.8"

To be considered a match, the strings must have exactly the same number of dashes in exactly the same positions.

Your method should throw an IllegalArgumentException if either string passed is null or the strings are different lengths.

Below are more examples of calls.

Call	Returns
sameDashes("abc","xyz")	true
sameDashes("abcd","xyz-")	false
sameDashes("--c-s-e","--1-4-3")	true
sameDashes("-1-2-3","1-2-3-")	false
sameDashes("c---a-t","c--a--t")	false
sameDashes("%*4ZY","%*4ZY")	true
sameDashes("----","----")	true
sameDashes("","")	true

You may not construct any structured objects other than Strings (no array, List, Scanner, etc.). You may not use any loops to solve this problem; you must use recursion. You may use any of the String methods listed on the cheat sheet.

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]->[3] q	p->[1]->[2] q->[3]	
p->[1]->[3] q->[2]	p->[1]->[2]->[3] q	
p->[1]->[2] q->[3]->[4]	p->[2]->[4] q->[3]->[1]	
p->[1]->[2]->[3] q->[4]->[5]	p->[2]->[1]->[4] q->[3]->[5]	

4. **Collections Programming, 20 points:** Write a method called computeTotalCredits that takes a map of student enrollments and a map of how many credits each course is worth, and returns a map that indicates how many credits each student is taking in total. The enrollments map uses student names as keys (strings) and has sets of course names as values (also strings). The credits map uses course names as keys (strings) and has the number of credits that course is worth as values (integers). The map you are to return should map student names to the total number of credits that student is enrolled in.

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

```
{"Porter"=["CSE143"], "Maria"=["CSE311", "PHIL120"],
 "Soham"=["PHIL120", "MATH308"], "Cherie"=["CSE143", "MATH308"],
 "Sea-Eun"=["CSE311", "MATH308", "PHIL120"], "Erik"=[]}
```

and a variable called credits might contain the following map:

```
{"AMATH301"=4, "CSE143"=5, "CSE311"=4, "MATH308"=3, "PHIL120"=5}
```

In this example, the enrollments map indicates that Maria is taking both CSE 311 and PHIL 120 and that Porter is only taking CSE 143. Also in this example, the credits map indicates that CSE 311 is a 4 credit class while MATH 308 is a 3 credit class. Suppose that the following call is made:

```
computeTotalCredits(enrollments, credits)
```

Given this call, the method would return a map whose keys are all the students in the enrollments and whose values are the total number of credits being taken by that student. Therefore, from this example this call would return the map

```
{"Cherie"=8, "Erik"=0, "Maria"=9, "Porter"=5, "Sea-Eun"=12, "Soham"=8}
```

Notice that because Maria is enrolled in CSE 311 (4 credits) and PHIL 120 (5 credits) that her value in the result is 9, while Porter's value is 5 because he is only enrolled in CSE 143 (5 credits).

The map you return should have keys ordered alphabetically. If there is a student enrolled in no classes, their corresponding value should be 0. If there are no students in the given map, you should return an empty map.

You may assume that the given maps are not null and you may assume none of the contents of the maps contain null values. You may also assume that for every class a student is enrolled in there is a corresponding entry for that class in the credits map. 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 sortPairs that takes a Stack of integers as a parameter and that sorts successive pairs of numbers starting at the bottom of the stack so that the smaller value appears at the bottom. For example, if the stack initially stores these values:

bottom [3, 8, 17, 9, 99, 9, 17, 8, 3, 1, 2, 3, 4, 14] top

your method should not switch the first pair (3, 8) since it is already sorted, while it should swap the second pair (17, 9) and the third pair (99, 9) since the larger number of the pair appears closer to the bottom of the stack. Applying this rule to each pair results in the following stack after sortPairs has been called:

bottom [3, 8, 9, 17, 9, 99, 8, 17, 1, 3, 2, 3, 4, 14] top

If there are an odd number of values in the stack, the value at the top of the stack is not moved. For example, if the original stack had stored:

bottom [3, 8, 17, 9, 99, 9, 17, 8, 3, 1, 2, 3, 4, 14, 42] top

It would again switch pairs of values, but the value at the top of the stack (42) would not be moved, yielding this sequence:

bottom [3, 8, 9, 17, 9, 99, 8, 17, 1, 3, 2, 3, 4, 14, 42] top

You may not make any assumptions about how many elements are in the stack.

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 (recall that you can't use the peek method or a foreach loop or iterator).

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

```
// Moves all elements from s to q
public static void s2q(Stack s, Queue q) { ... }
// Moves all elements from q to s
public static void q2s(Queue q, Stack s) { ... }
```

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

6. **ArrayIntList Programming, 10 points:** Write a method called mirror that doubles the size of a list of integers by appending the mirror image of the original sequence to the end of the list. The mirror image is the same sequence of values in reverse order. For example, if a variable called list stores this sequence of values:

```
[1, 3, 2, 7]
```

and you make the following call:

```
list.mirror();
```

then it should store the following values after the call:

```
[1, 3, 2, 7, 7, 2, 3, 1]
```

Notice that it has been doubled in size by having the original sequence appearing in reverse order at the end of the 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>
}
```

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). You may assume that the array has sufficient capacity to store the new values.

^_^ 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

`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
`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
`isEmpty()`	`true` if this String is the empty 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

`Math` Methods (3.2)

`abs(`**`x`**`)`	returns the absolute value of `x`
`max(`**`x, y`**`)`	returns the larger of `x` and `y`
`min(`**`x, y`**`)`	returns the smaller of `x` and `y`
`pow(`**`x, y`**`)`	returns the value of `x` to the `y` power
`random()`	returns a random number between `0.0` and `1.0`
`round(`**`x`**`)`	returns `x` rounded to the nearest integer