CSE143 Sample Midterm
Winter 2019

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 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 value is returned.

Method Call	Value Returned
mystery(42, 0)	_____
mystery(5, 25)	_____
mystery(752, 792)	_____
mystery(3092, 5028)	_____
mystery(61903, 1930)	_____

2. **Recursive Programming, 15 points:** Write a recursive method called groupChars that takes a string as a parameter and that returns a new string obtained by inserting parentheses and brackets so as to group the characters. For strings with only one or two characters, the characters should be surrounded by square brackets. For example, groupChars("in") should return "[in]" and groupChars("a") should return "[a]". For strings with more than 2 characters, parentheses should be inserted that surround and separate individual characters with the center-most characters surrounded by square brackets. The table below includes more examples.

Method Call	Value Returned
groupChars("the")	"(t[h]e)"
groupChars("rain")	"(r[ai]n)"
groupChars("in")	"[in]"
groupChars("Spain")	"(S(p[a]i)n)"
groupChars("falls")	"(f(a[l]l)s)"
groupChars("mainly")	"(m(a[in]l)y)"
groupChars("recursively!")	"(r(e(c(u(r[si]v)e)l)y)!)"
groupChars("")	"*"

Notice that the method might be passed an empty string, in which case it returns a string composed of a single asterisk. You are not allowed to construct any structured objects to solve this problem other than strings (no array, ArrayList, StringBuilder, Scanner, etc) and you may not use a while loop, for loop or do/while loop to solve this problem; you must use recursion. You may use only the string methods included 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] q->[3]	p->[1] q->[3]->[2]	
  p q->[1]->[2]->[3]	 p->[1] q->[2]->[3]	
  p->[1]->[2]->[3] q->[4]	 p->[3]->[1] q->[2]->[4]	
  p->[1] q->[2]->[3]->[4]->[5]	 p->[5]->[1]->[3] q->[4]->[2]	

4. **Collections Programming, 20 points:** Write a method named cancelCourse that accepts two parameters

1. A String representing the name of a course which is being canceled
2. A Map in which keys represent student names and values represent the set of courses in each students' schedule

Your method should remove the specified course from all students' schedules. It should return the set of names of the students whose schedules have changed. The set returned should be sorted alphabetically.

For example, given the following map named schedules:

```
{Chin=[SOC 345, CSE 143, LING 400 ],
 Kevin=[ENGL 131, SOC 345, LING 400],
 Yael=[ENGL 131, CSE 143, EE 215, LING 400]}
```

The call cancelCourse("CSE 143", schedules) returns [Chin, Yael] because Chin and Yael were both previously registered for CSE 143 while Kevin was not.

After this call, schedules should contain

```
{Chin=[SOC 345, LING 400 ],
 Kevin=[ENGL 131, SOC 345, LING 400],
 Yael=[ENGL 131, EE 215, LING 400]}
```

Your method should only remove courses that match the parameter exactly, including case. For example, "cse 143" is not the same course as "CSE 143".

You may assume the map passed in to your method is not null and that it does not contain null values.

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 mirrorCollapse that takes a stack of integers as a parameter and that collapses the stack by combining pairs of values whose positions are a mirror image of each other. For example, suppose a variable s stores these values:

```
bottom [1, 2, 3, 40, 50, 60] top
        ^  ^  ^  ^   ^   ^
        |  |  +--+   |   |
        |  +---------+   |
        +----------------+
         mirror positions
```

The pairs of values in mirror image positions are the first and last (1 and 60), the second and fifth (2 and 50), and the third and fourth (3 and 40).

The method should add the first value to its mirror image, add the second value to its mirror image, add the third value to its mirror image, and so on. Thus, if we make the following call:

```
mirrorCollapse(s);
```

the stack should store the following values after the call:

```
bottom [43, 52, 61] top
```

If there is a value in the middle that has no mirror image, then it should not be altered. For example, if the stack had instead stored these values:

```
bottom [1, 2, 3, 4, 50, 60, 70] top
```

then it should store the following values after the method is called:

```
bottom [4, 53, 62, 71] top
```

Notice that the value 4 that was in the middle is unchanged. This example uses values with a regular pattern to make it easier to understand what is going on, but you should not assume anything about the sequence. For example, if s instead stored this sequence:

```
bottom [7, 1, 4, 18, 9, 23, 0, -5, 12] top
```

then after the method is called, it would store this sequence:

```
bottom [9, 41, 4, -4, 19] top
```

Your method should not change the stack if it has fewer than two values.

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 may assume these helper methods have been defined

```
// 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 fromCounts that constructs and returns a new ArrayIntList of values given an existing ArrayIntList of counts. Assume that the ArrayIntList that is called stores a sequence of integer pairs that each indicate a count and a number. For example, suppose that an ArrayIntList called list stores the following sequence of values:

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

This sequence indicates that you have 5 occurrences of 2, followed by two occurrences of -5, followed by 4 occurrences of 3, and so on. If we make the following call, storing the value returned in a variable called list2:

```
ArrayIntList list2 = list.fromCounts();
```

Then the variable list2 should store the following sequence of values:

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

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 may assume that the ArrayIntList that is called stores a legal sequence of pairs, which means it will always have an even size, and that the zero-argument constructor for ArrayIntList will construct an array of sufficient capacity to store the result. If the sequence of pairs is empty, the result should be an empty list.

You may call the ArrayIntList constructor, but otherwise you may not call any other methods of the ArrayIntList class to solve this problem. You are not allowed to define any auxiliary data structures other than the new ArrayIntList you are constructing (no array, String, ArrayList, etc), and you may not change the original 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

`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