CSE 143, Winter 2009
Midterm Exam
Wednesday February 18, 2009

<u>Personal Information:</u> (1 point)

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/notes. 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 <u>only</u> 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**
	Personal Info / Conduct		1
1	Stacks and Queues		19
2	Sets and Maps		15
3	Linked List Nodes		15
4	Linked List Programming		20
5	Recursive Tracing		15
6	Recursive Programming		15
TOTAL	**Total Points**		**100**

1. Stacks and Queues

Write a method `mirrorHalves` that accepts a queue of integers as a parameter and replaces each half of that queue with itself plus a mirrored version of itself (the same elements in the opposite order). For example, suppose a variable `q` stores the following elements (each half is underlined for emphasis):

```
front [10, 50, 19, 54, 30, 67] back
```

After a call of `mirrorHalves(q);`, the queue would store the following elements (new elements in bold):

```
front [10, 50, 19, 19, 50, 10, 54, 30, 67, 67, 30, 54] back
```

If your method is passed an empty queue, the result should be an empty queue. If your method is passed a `null` queue or one whose size is not even, your method should throw an `IllegalArgumentException`.

You may use **one stack or one queue** (but not both) 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 may not use recursion to solve this problem. For full credit your code must run in O(*n*) time where *n* is the number of elements of the original queue. Use the `Queue` interface and `Stack`/`LinkedList` classes from lecture.

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) { ... }
```

2. Sets and Maps

Write a method `rarestAge` that accepts as a parameter a map from students' names (strings) to their ages (integers), and returns the *least* frequently occurring age. Consider a map variable `m` containing the following key/value pairs:

```
{Alyssa=22, Char=25, Dan=25, Jeff=20, Kasey=20, Kim=20, Mogran=25, Ryan=25, Stef=22}
```

Three people are age 20 (Jeff, Kasey, and Kim), two people are age 22 (Alyssa and Stef), and four people are age 25 (Char, Dan, Mogran, and Ryan). So a call of `rarestAge(m)` returns `22` because only two people are that age.

If there is a tie (two or more rarest ages that occur the same number of times), return the youngest age among them. For example, if we added another pair of `Kelly=22` to the map above, there would now be a tie of three people of age 20 (Jeff, Kasey, Kim) and three people of age 22 (Alyssa, Kelly, Stef). So a call of `rarestAge(m)` would now return `20` because 20 is the smaller of the rarest values.

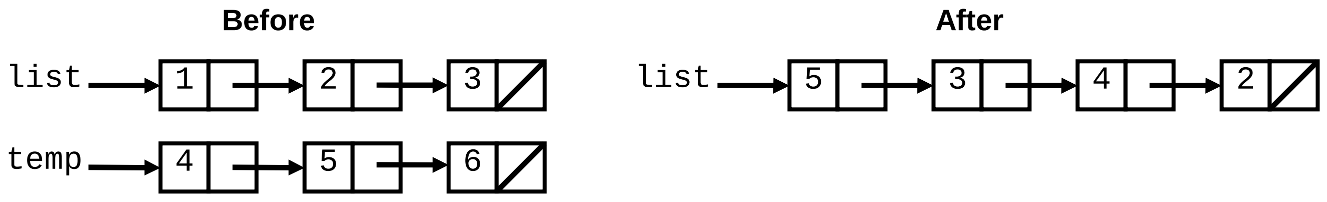
If the map passed to your method is `null` or empty, your method should throw an `IllegalArgumentException`. You may assume that no key or value stored in the map is `null`. Otherwise you should not make any assumptions about the number of key/value pairs in the map or the range of possible ages that could be in the map.

You may create **one new set or map** as auxiliary storage to solve this problem. You can have as many simple variables as you like. You should not modify the contents of the map passed to your method. For full credit your code must run in less than O(n^2) time where n is the number of pairs in the map.

3. Linked List Nodes

Write the code that will turn the Before picture into the After picture below by modifying links between nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but you are NOT allowed to change any node's `data` field value. You also should not create new `ListNode` objects, but you may create a single `ListNode` variable to refer to any existing node if you like. If a variable does not appear in the After picture, it doesn't matter what value it has after the changes are made.

To help maximize partial credit in case you make mistakes, we suggest that you include optional comments with your code that describe the links you are trying to change, as shown in Section 7's solution code.



Assume that you are using the `ListNode` class as defined in lecture and section:

```
public class ListNode {
    public int data;       // data stored in this node
    public ListNode next;  // a link to the next node in the list

    public ListNode() { ... }
    public ListNode(int data) { ... }
    public ListNode(int data, ListNode next) { ... }
}
```

4. Linked List Programming

Write a method `compress` that could be added to the `LinkedIntList` class, that accepts an integer *n* representing a "compression factor" and replaces every *n* elements with a single element whose `data` value is the sum of those *n* nodes. Suppose a `LinkedIntList` variable named `list` stores the following values:

```
[2, 4, 18, 1, 30, -4, 5, 58, 21, 13, 19, 27]
```

If you made the call of `list.compress(2);`, the list would replace every two elements with a single element (2 + 4 = **6**, 18 + 1 = **19**, 30 + (-4) = **26**, ...), storing the following elements:

```
[6, 19, 26, 63, 34, 46]
```

If you then followed this with a second call of `list.compress(3);`, the list would replace every three elements with a single element (6 + 19 + 26 = **51**, 63 + 34 + 46 = **143**), storing the following elements:

```
[51, 143]
```

If the list's size is not an even multiple of *n*, whatever elements are left over at the end are compressed into one node. For example, the original list on this page contains 12 elements, so if you made a call on it of `list.compress(5);`, the list would compress every five elements, (2 + 4 + 18 + 1 + 30 = **55**, -4 + 5 + 58 + 21 + 13 = **93**), with the last two leftover elements compressing into a final third element (19 + 27 = **46**), resulting in the following list:

```
[55, 93, 46]
```

If *n* is greater than or equal to the list size, the entire list compresses into a single element. If the list is empty, the result after the call is empty regardless of what factor *n* is passed. You may assume that the value passed for *n* is ≥ 1.

For full credit, you may not create any new `ListNode` objects, though you may have as many `ListNode` variables as you like. For full credit, your solution must also run in O(*n*) time. Assume that you are adding this method to the `LinkedIntList` class below. You may not call any other methods of the class.

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

5. Recursive Tracing

For each call to the following recursive method, indicate what output is produced:

```
public void mystery(int n) {
    if (n < 0) {
        System.out.print("-");
        mystery(-n);
    } else if (n < 10) {
        System.out.println(n);
    } else {
        int two = n % 100;
        System.out.print(two / 10);
        System.out.print(two % 10);
        mystery(n / 100);
    }
}
```

Call	Output
`mystery(7);`	
`mystery(825);`	
`mystery(38947);`	
`mystery(612305);`	
`mystery(-12345678);`	

6. Recursive Programming

Write a recursive method `isReverse` that accepts two strings as a parameter and returns `true` if the two strings contain the same sequence of characters as each other but in the opposite order (ignoring capitalization), and `false` otherwise. For example, the string `"hello"` backwards is `"olleh"`, so a call of `isReverse("hello", "olleh")` would return `true`. Since the method is case-insensitive, you would also get a `true` result from a call of `isReverse("Hello", "oLLEh")`. The empty string, as well as any one-letter string, is considered to be its own reverse. The string could contain characters other than letters, such as numbers, spaces, or other punctuation; you should treat these like any other character. The key aspect is that the first string has the same sequence of characters as the second string, but in the opposite order, ignoring case. The table below shows more examples:

Call	Value Returned
`isReverse("CSE143", "341esc")`	`true`
`isReverse("Madam", "MaDAm")`	`true`
`isReverse("Q", "Q")`	`true`
`isReverse("", "")`	`true`
`isReverse("e via n", "N aIv E")`	`true`
`isReverse("Go! Go", "OG !OG")`	`true`
`isReverse("Obama", "McCain")`	`false`
`isReverse("banana", "nanaba")`	`false`
`isReverse("hello!!", "olleh")`	`false`
`isReverse("", "x")`	`false`
`isReverse("madam I", "i m adam")`	`false`
`isReverse("ok", "oko")`	`false`

You may assume that the strings passed are not `null`. You are not allowed to construct any structured objects other than `Strings` (no array, `List`, `Scanner`, etc.) and you may not use any loops to solve this problem; you must use recursion. If you like, you may declare other methods to help you solve this problem, subject to the previous rules.