CSE 143, Summer 2012
Midterm Exam
Monday, July 23, 2012

Personal Information:

Name: ____

Section: __________________ TA: __________________

Student ID #: __________________

- You have 60 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. You may not use anything other than the class textbook. 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 only 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 write your answer on scratch paper, please clearly write your name on every sheet and write a note on the original sheet directing the grader to the scratch paper. We are not responsible for lost scratch paper or for answers on scratch paper that are not seen by the grader due to poor marking.
- 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!

Problem	Description	Earned	Max
1	ArrayList Mystery		15
2	ArrayList Programming		15
3	Stacks and Queues		15
4	Linked Nodes		10
5	LinkedList Programming		15
6	Recursive Tracing		15
7	Recursive Programming		15
TOTAL	**Total Points**		**100**

1. ArrayList Mystery.
Consider the following method:

```
public static void mystery1 (ArrayList<Integer> list) {
      for (int i = 1; i < list.size(); i++) {
            if (list.get(i-1) < list.get(i)) {
                  list.remove(i);
                  list.add(i, list.get(i-1));
            }
      }
}
```

Write the final contents of the `ArrayList` after the method is called with the initial `ArrayList` contents on the left:

List	**Output**
(a)	
`[1, 2, 3, 4]`	________[1, 1, 1, 1]____________________
(b)	
`[5, -1, 3, 2, -2, 4, 2, 1]`	________[5, -1, -1, -1, -2, -2, -2, -2]_______
(c)	
`[5, 5, 5, 0, 5, 5, 5]`	________[5, 5, 5, 0, 0, 0, 0]______________
(d)	
`[4, 3, 2, 1, 0, 0, -16, -1]`	________[4, 3, 2, 1, 0, 0, -16, -16]_________

2. ArrayList Programming.

Write a method `makeConsecutiveByN` that could be added to the `ArrayIntList` class that takes an integer *n* as a parameter, removes any list element that does not differ from the previous element by exactly *n*. For example, both 8 and 2 differ from 5 by exactly 3. Your method should return `true` if the list was changed and `false` if not. For example, suppose `elementData` stores the following element values: `[1, 3, -6, 1]`. We need to compare every set of pairs. If we remove an element we need to compare the next element in the list to the last non-removed element. If n were 2 and we made a call on the previous list we would compare 1 and 3, keep 3 and then compare 3 and -6. We would discard -6 and so then compare 3 and the last 1. Our call would return `true.` You may not use any other arrays, lists, or other data structures to help you solve this problem, though you can create as many simple variables as you like.

Call	**Result**
`elementData = [1, 3, 5, 3, 5, 3, 1]` `n = 2`	`elementData = [1, 3, 5, 3, 5, 3, 1]` `returns false`
`elementData = [1, 3, 5, 5, 5, 3, 1]` `n = 2`	`elementData = [1, 3, 5, 3, 1]` `returns true`
`elementData = [1, 3, -6, -3, 5, 3, 1]` `n = 3`	`elementData = [1]` `returns true`
`elementData = [24,14,7,4,-6,25,-16,26,-6]` `n = 10`	`elementData = [24, 14, 4, -6, -16, -6]` `returns true`

Solution:

```
public boolean makeConsecutiveByN (int n) {
      boolean changed = false;
      if (size > 0) {
            int current = elementData[0];
            for (int i = 1; i < size; i++) {
                  if (current+n != elementData[i] && current-n != elementData[i]) {
                        remove(i);
                        i--;
                        changed = true;
                  }
                  current = elementData[i];
            }
      }
      return changed;
}
```

3. Stack and Queue Programming.

Write a method `mirror` that accepts a stack of integers as a parameter and replaces the stack contents with itself plus a mirrored version of itself (the same elements in the opposite order). For example, suppose a variable s stores the following elements:

```
bottom [10, 50, 19, 54, 30, 67] top
```

After a call of `mirror(s);`, the stack would store the following elements (underlined for emphasis):

```
bottom [10, 50, 19, 54, 30, 67, 67, 30, 54, 19, 50, 10] top
```

Note that the mirrored version is added on to the top of what was originally in the stack. The bottom half of the stack contains the original numbers in the same order. If your method is passed an empty stack, the result should be an empty stack. If your method is passed a `null` stack, 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 stack. 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) {
      while (!s.isEmpty()) {
            q.add(s.pop());
      }                               // Transfers the entire contents
}                                     // of stack s to queue q

public static void q2s(Queue<Integer> q, Stack<Integer> s) {
      while (!q.isEmpty()) {
            s.push(q.remove());
      }                               // Transfers the entire contents
}                                     // of queue q to stack s
```

Solution:

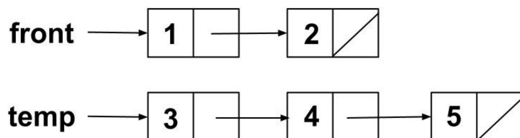
```
public static void mirror (Stack<Integer> s) {
            if (s == null)
                  throw new IllegalArgumentException();
            Queue<Integer> q = new LinkedList<Integer>();

            while (!s.isEmpty()) {
                  q.add(s.pop());
            }
            for (int i = 0; i < q.size(); i++) {
                  int n = q.remove();
                  q.add(n);
                  s.push(n);
            }
            int size = q.size();          // can be the size of either,
            while (!s.isEmpty()) {        // they are the same
                  q.add(s.pop());
            }
            for(int i = 0; i < size; i++) {
                  q.add(q.remove());
            }
            while (!q.isEmpty()) {
                  s.push(q.remove());
            }
}
```

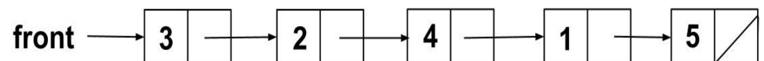
4. Linked Nodes.
Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown and/or creating new nodes as needed. There may be more than one way to write the code, but you are NOT allowed to change any existing node's `data` field value. You also should not create new `ListNode` objects unless necessary to add new values to the chain, 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.

Before



After



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) { ... }

}
```

Solution:

```
ListNode current = front.next;
front.next = temp.next.next;       // 1 → 5
temp.next.next = front;            // 4 → 1
current.next = temp.next;          // 2 → 4
temp.next = current;               // 3 → 2
front = temp;
```

5. LinkedList Programming.
Write a method `stutterTwo` that could be added to the `LinkedIntList` class that doubles the size of the list by replacing every pair of neighboring integers *a,b* with two consecutive copies of that pair, *a,b,a,b* (a total of four integers come from the original two). Suppose a `LinkedIntList` variable named list stores the following elements from front to back:

```
[18, 4, 27, 9, 54, 5, 63]
```

If you made the call of `list.stutterTwo();`, the list would then store the elements in this order:

```
[18, 4, 18, 4, 27, 9, 27, 9, 54, 5, 54, 5, 63]
```

If the list has an odd number of elements the last element should not be stuttered. If the list is empty, its contents should not be modified.

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

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

Solution:

```
      public void stutterTwo() {
            ListNode current = front;
            while (current != null && current.next != null) {
                  ListNode copy = new ListNode (current.data,
                              new ListNode (current.next.data, current.next.next));
                  current.next.next = copy;
                  current = current.next.next.next.next;
            }
      }
```

6. Recursive Tracing.
For each call to the following method, indicate what value is returned:

```
public static String mystery (String s, char c) {
      if (s.length() == 0) {
            return s;
      } else if (s.charAt(s.length() - 1) == c) {
            return c + mystery(s.substring(0, s.length() - 1), c);

      } else {
            int len =  s.length() - 1;
            return mystery(s.substring(0, len), c) + s.charAt(len);
      }
}
```

Call	**Output**
mystery("sce", "c");	cse
mystery("static", "t");	ttsaic
mystery("banana", "a");	aaabnn
mystery("java", "j");	java
mystery("ALL", "L");	LLA

7. Recursive Programming.

Write a recursive method `parenthesize` that accepts an int *n* as a parameter and prints out the numbers 1 through *n* inclusive in a particular pattern that looks like a set of mathematical additions wrapped in parentheses. The order of the numbers should begin with all of the evens in downward order, followed by all of the odds upward from 1. Each time a number is added to the pattern, a new set of parentheses and a + sign are added too. You may assume that the number passed to your method is greater or equal to 0. Look at the pattern in the calls below to see the print format.

Call	Output
`parenthesize(0);`	`0`
`parenthesize(1);`	`1`
`parenthesize(2);`	`(2 + 1)`
`parenthesize(3);`	`((2 + 1) + 3)`
`parenthesize(4);`	`(4 + ((2 + 1) + 3))`
`parenthesize(5);`	`((4 + ((2 + 1) + 3)) + 5)`
`parenthesize(6);`	`(6 + ((4 + ((2 + 1) + 3)) + 5))`
`parenthesize(7);`	`((6 + ((4 + ((2 + 1) + 3)) + 5)) + 7)`
`parenthesize(8);`	`(8 + ((6 + ((4 + ((2 + 1) + 3)) + 5)) + 7))`
`parenthesize(9);`	`((8 + ((6 + ((4 + ((2 + 1) + 3)) + 5)) + 7)) + 9)`

You are not allowed to construct any structure objects (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.

Solution:

```
public static void parenthesize( int n) {
      if (n/2 == 0 ) {
            System.out.print(n);
      } else if (n % 2 == 0) {
            System.out.print("(" + n + " + ");
            parenthesize(n - 1);
            System.out.print(")");
      } else {
            System.out.print("(");
            parenthesize(n - 1);
            System.out.print(" + " + n + ")");
      }
}
```