CIS 110 Spring 2014 — Introduction to Computer Programming
12 May 2014 — Final Exam
Answer Key

0.) THE EASY ONE (1 point total)

Check coversheet for name, recitation #, PennKey, and signature.

1.) MULTIPLE CHOICE (10 points total)

1.1) (4 points) For each scenario below, choose whether it would be better to store the data in an array, an `ArrayList`, or a `LinkedList`. Circle one answer for each scenario.

(a) The data rarely changes, and the program should use the least amount of memory possible:
`array` `ArrayList` `LinkedList`

(b) The amount of data will vary over time, but memory usage should **always** be minimal:
`array` `ArrayList` **`LinkedList`**

(c) The order of the data is constantly changing:
`array` `ArrayList` **`LinkedList`**

(d) The data is PennIDs of students, updated daily, and repeatedly searched using binary search:
`array` **`ArrayList`** `LinkedList`

1.2) (2 points) Which algorithm for sorting a large array of integers is the fastest?

(a) Insertion sort

(b) **Merge sort**

(c) Selection sort

(d) Bubble sort

(e) They are all equally fast

1.3) (2 points) Which of the following expressions is true? (Circle only one)
(Recall the following bitwise operators: `&` (and), `|` (or), `^` (exclusive or).)

(a) `(1 ^ 0) & 1 == 1`

(b) `(0 & 1) == 1`

(c) `1 ^ (0 ^ 1) == 1`

(d) `!((0 | 1) == 1)`

(e) `(1 | (1 ^ (1 & 0))) != 1`

1.4) (2 points) In TOY, the program counter represents:

(a) The number of available memory slots

(b) The number of lines of code in the program

(c) The amount of time the program has been running

(d) **The memory address of the instruction that is currently being executed**

(e) The number of CPU clock tics since the computer last rebooted

2.) METHOD OVERLOADING (8 points total)

Your start-up's backers have hired a seasoned executive to run you company. He has proposed the following API for the `add()` methods for a `LinkedList` of integers, but he has forgotten to consider Java's rules for method overloading:

(A) `public void add(int x)`
(B) `public void add(int index, int x)`
(C) `public void add(int x, int index)`
(D) `public boolean add(int x, int index)`
(E) `public void add(int x, boolean duplicates)`
(F) `private boolean add(int x)`

2.1) (5 points) Circle `Yes` or `No` to indicate whether each of the pairs of methods listed below could both be included in the same Java API:

(A) and (B): **Yes** No
(A) and (F): Yes **No**
(B) and (C): Yes **No**
(C) and (D): Yes **No**
(D) and (E): **Yes** No

2.2) (3 points) After some debate, you decide to ignore your seasoned executive, and you condense the API to the following:

```
public LinkedList()
public void add(int x)
public void add(int x, int index)
public int get(int index)
public int size()
public void remove(int index)
```

One hallmark of good programming is building upon other methods that you've already written. Provide an implementation of `add(int x)` that relies solely on the other methods from the condensed API above. Do not access any fields of the linked list directly.

```
public void add(int x) {
    add(x, size());
}
```

3.) LISTS AND METHODS AND COMPLEXITY, O(MY)! (12 points total)

For each list operation, circle the computational complexity for each data structure. Make certain that you have circled one complexity in both columns within each row. The first row has been completed for you. For `ArrayLists`, you may assume that the underlying array does not need to be resized. You should also assume efficient implementations for all methods and data structures.

	`ArrayList`		`DoublyLinkedList`	
Get the element at the head of the list	[$O(1)$]	$O(n)$	[$O(1)$]	$O(n)$
Construct an empty list	[$O(1)$]	$O(n)$	[$O(1)$]	$O(n)$
Get the size of the list	[$O(1)$]	$O(n)$	[$O(1)$]	$O(n)$
Add an element to the head of the list	$O(1)$	[$O(n)$]	[$O(1)$]	$O(n)$
Add an element to the tail of the list	[$O(1)$]	$O(n)$	[$O(1)$]	$O(n)$
Get the element at index i in the list	[$O(1)$]	$O(n)$	$O(1)$	[$O(n)$]
Remove the element at index `0`	$O(1)$	[$O(n)$]	[$O(1)$]	$O(n)$

4.) `REMOVE(REMOVE(REMOVE(REMOVE(...))))` (9 points total)

Fill in the blanks in the partial recursive `LinkedList` implementation below to complete the class. Each of your answers should be one line of code or less. You may assume that all other methods of the class have already been completed.

```
public RecursiveLinkedListOfStrings {
    private Node head;
    private int size;

    private class Node {
        public String value;
        public Node next;
        public Node(String v, Node n) {  value = v;  next = n;  }
    }

    ...   // other LinkedList methods would go here

    /* Removes the first instance of the specified value from the list */
    public void remove(String value) {
        ________________________________________________ ;   /* LINE A */
    }

    /* Removes the first node containing s from the sublist rooted at n.
     * n may be null if we are at the end of the list.
     * Returns the sublist, with the specified value removed
     */
    private Node recursiveRemove(Node n, String s) {
        if (n == null) {                // reached the end of the list
            return ______________________________________ ;   /* LINE B */
        } else if (n.value.equals(s)) {  // found a matching node
            size--;
            return ______________________________________ ;   /* LINE C */
        }
        n.next = recursiveRemove(n.next, s);
        return n;
     }
}
```

Write your answers in the spaces below:

(a) Line A: `head = recursiveRemove(head, value);`

(b) Line B: `n`

(c) Line C: `n.next`

5.) FUN-SIZE STACK PACK (15 points total)

Write a `Stack` class that conforms to the following API:

```
public class Stack
-------------------------------------------------------------------------
public Stack()              // create an empty stack
public void push(String s)  // push a new element on the stack
public String pop()         // remove and return the top element
                            //   return null if the stack is empty
public String top()         // peek at the top element without removing it
                            //   return null if empty
public boolean isEmpty()    // return whether the stack is empty
```

Make certain that all of your methods take $O(1)$ time. You do not need to comment your code, but you may if you wish to clarify your implementation.

You must write your `Stack` class **without** defining a separate *Node* class. Instead you must build it using an existing `LinkedList` class. In other words, your `Stack` object will contain a `LinkedList` object as a field, and will store all data in this `LinkedList`. Your `Stack` class must rely on the `LinkedList`'s methods, and must not create or manipulate individual nodes directly.

```
public class Stack {
    LinkedList data;

    public Stack() {  data = new LinkedList(); }

    public void push(String s) { data.add(s, 0); }

    public String pop() {
        if (isEmpty()) return null;
        return data.remove(0);
    }

    public String top() {
        if (isEmpty()) return null;
        return data.get(0);
    }

    public int isEmpty() { return data.size() == 0; }
}
```

6.) ODE TO TOY (11 points total)

Your friendly TAs wrote the following TOY program for fun during our end-of-semester staff picnic. Unfortunately, they did a lousy job. They didn't write proper comments or document what the program does, and there seems to be a serious bug that makes program behave unpredictably. At least the auto-generated assembly comments are correct.

```
10: 8AFF   read R[A]
11: 8BFF   read R[B]
12: 21AB   R[1] <- R[A] - R[B]
13: 9BFF   write R[B]
14: C121   if (R[1] == 0) goto 21
15: 7373   R[3] <- 0073
16: 4333   R[3] <- R[3] ^ R[3]
17: 93FF   write R[3]
18: 1AAA   R[A] <- R[A] + R[A]
19: 211B   R[1] <- R[1] - R[B]
20: C014   if (R[0] == 0) goto 14
21: 8CFF   read R[C]
22: 3CBC   R[C] <- R[B] & R[C]
23: 9CFF   write R[C]
24: 0000   halt
```

6.1) (2 points) In **20 words or less**, describe the bug that makes program behave unpredictably.
The program jumps from memory address `0x19` to `0x20` instead of to `0x1A`.
Note: Identifying the infinite loop in the program did not receive credit because it does not make the program behave unpredictably.

6.2) (3 points) Correct the bug you described above. Write out the corrected line of code for each line that changes. You do not need to write out the lines that do not change, and you do not need to write out the auto-generated assembly comments (or any other comments).

```
14: C11B  // correct the branch target

1A: C014  // renumber the last five instructions
1B: 8CFF
1C: 3CBC
1D: 9CFF
1E: 0000
```

6.3) (6 points) Given the following input, print the output you would receive once the bug has been fixed. If you believe the original program listing is correct, give the output you think it will produce. You may write your answers in base 10 (decimal) or base 16 (hexadecimal).

(a) `1 1 8`

`1 0` *Note: No credit for answer consistent with a mis-corrected program.*

(b) `6 2 8`

`2 0 0 0`

(c) `5 2 9`

`2 0 0 ...` (0s forever)

7.) A RAY IF I DO, A RAY IF I DON'T (10 points total)

The following function takes a string and a length as arguments, and returns an array of the specified length containing the string's characters.

```
1 public static char[] arrayify(String str, int length) {
2     char[] chars = new char[length];
3     for (int i = 0; i < str.length(); i++)
4         chars[i] = str.charAt(i);
5     return chars;
6 }
```

For example, `arrayify("110 is a ray of sun.", 20)` would return the 20-element array:

```
{ '1', '1', '0', ' ', 'i', 's', ' ', 'a', ' ', 'r', 'a', 'y', ' ', 'o', 'f', ' ', 's', 'u', 'n', '.' }
```

7.1) (5 points) For each of the calls to `arrayify` in the left column, write the <u>one</u> letter from the right column of the exception it would cause (or indicating there would be none):

D	`arrayify("hi", 2);`	(a) `java.lang.ArrayIndexOutOfBoundsException`
A	`arrayify("hi", 1);`	(b) `java.lang.NullPointerException`
D	`arrayify("hi", 5);`	(c) `java.lang.IllegalArgumentException`
B	`arrayify(null, 0);`	(d) This statement will not cause an exception.
B	`arrayify(null, 1);`	

7.2) (5 points) Write a version of `arrayify()` that returns `null` instead of throwing exceptions. The return value should match the version of `arrayify()` above, except when an exception would have been thrown. If your implementation reuses lines of code that already appear above, write "**Line X**" instead of rewriting that line of code. For instance, write "**Line 5**" instead of "`return chars;`"

```
public static char[] arrayify(String str, int length) {
    if (str == null) return null;
    if (length < str.length()) return null;
    Line 2
    Line 3
    Line 4
    Line 5
}

--- OR ---

public static char[] arrayify(String str, int length) {
    if (str == null) return null;
    if (length < str.length()) return null;
    char[] chars = new char[length];
    for (int i = 0; i < str.length(); i++)
            chars[i] = str.charAt(0);
    return chars;
}
```

8.) BAIT AND SWITCH (8 points total)

The following code models fishers who fish for fish in the sea. They aren't very good though, and sometimes they hook each other instead. This code is completely correct (and tested), except for one line in the `Fish` class. Sadly, the fishers aren't much better at programming than they are at fishing.

8.1) (3 points) Find and correct the single line of code in the `Fish` class that behaves incorrectly. You only need to write the corrected line of code below; you do not need to write any of the surrounding code or explain the error.

```
this.name = name;
```

8.2) (5 points) The following program models a "catch-and-release" fishing expedition. Anything that is caught is immediately released and can be caught again. Fill in the blanks below in the program's output. Assume the random values that are assigned to the variable `randomFish` are, in order: **3, 2, 3, 4, 1, 2, 3, 4, 4, 1**.

Carrie	caught	Stanley
Rod	caught	Wanda
Carrie	caught	Bernie
Rod	caught	nothing
Carrie	caught	Rod
Rod	caught	Wanda
Carrie	caught	Luke
Rod	caught	nothing
Carrie	caught	nothing
Rod	caught	Rod

Have a great summer!