CIS 110 — Introduction to Computer Programming

7 May 2012 — Final Exam

Name: _____

Recitation # (e.g. 201): _____

Pennkey (e.g. bjbrown): _____

My signature below certifies that I have complied with the University of Pennsylvania's Code of Academic Integrity in completing this examination.

_____ _____

Signature Date

Scores:

1		1
2		5
3		9
4		12
5		13
6		20
7		30
Total:		90

CIS 110 Final Instructions

- You have 110 minutes to finish this exam. Time will begin when called by a proctor and end precisely 110 minutes after that time. If you continue writing after the time is called, you will receive a zero for the exam.
- This exam is *closed-book, closed-notes, and closed-computational devices*. Except where noted, you can assume that code included in the question is correct and use it as a reference for Java syntax.
- This exam is long. If you get stuck part way through a problem, it may be to your advantage to go on to another problem and come back later if you have time.
- When writing code, the only abbreviations you may use are for `System.out.println`, `System.out.print`, and `System.out.printf` as follows:

  $$\text{System.out.println} \longrightarrow \text{S.O.PLN}$$
  $$\text{System.out.print} \longrightarrow \text{S.O.P}$$
  $$\text{System.out.printf} \longrightarrow \text{S.O.PF}$$

  Otherwise all code must be written out as normal, including all curly braces and semicolons.
- Please do not separate the pages of the exam. If a page becomes loose, write your name on it and use the provided staplers to reattach the sheet when you turn in your exam so that we don't lose it.
- If you require extra paper, please use the backs of the exam pages or the extra sheet(s) of paper provided at the end of the exam. Clearly indicate on the question page where the graders can find the remainder of your work (e.g. "back of page" or "on extra sheet"). Staple an extra sheets you use to the back of your exam when you turn it in using the provided staplers.
- If you have any questions, please raise your hand and an exam proctor will come to answer them.
- When you turn in your exam, you will be required to show ID. If you forgot to bring your ID, please talk to an exam proctor immediately.

Good luck, have fun!

TOY Reference Card

INSTRUCTION FORMATS

```
            | . . . . | . . . . | . . . . | . . . .|
  Format 1: | opcode  |    d    |    s    |    t   |  (0-6, A-B)
  Format 2: | opcode  |    d    |       addr       |  (7-9, C-F)
```

ARITHMETIC and LOGICAL operations

```
    1: add              R[d] <- R[s] +  R[t]
    2: subtract         R[d] <- R[s] -  R[t]
    3: and              R[d] <- R[s] &  R[t]
    4: xor              R[d] <- R[s] ^  R[t]
    5: shift left       R[d] <- R[s] << R[t]
    6: shift right      R[d] <- R[s] >> R[t]
```

TRANSFER between registers and memory

```
    7: load address     R[d] <- addr
    8: load             R[d] <- mem[addr]
    9: store            mem[addr] <- R[d]
    A: load indirect    R[d] <- mem[R[t]]
    B: store indirect   mem[R[t]] <- R[d]
```

CONTROL

```
    0: halt             halt
    C: branch zero      if (R[d] == 0) pc <- addr
    D: branch positive  if (R[d] >  0) pc <- addr
    E: jump register    pc <- R[d]
    F: jump and link    R[d] <- pc; pc <- addr
```

Register 0 always reads 0.
Loads from mem[FF] come from stdin.
Stores to mem[FF] go to stdout.

Miscellaneous

1. (1 points)

   (a) Write your name, recitation number, and PennKey (username) on the front of the exam.

   (b) Sign the certification that you comply with the Penn Academic Integrity Code

Find the Bugs

2. (5 points) Give five bugs in the following code. The line numbers are included for your convenience and are not part of the program. (Hint: Think about what the program is trying to do to find bugs in the logic.)

```
1:  int[][] myarray = int[5][4];
2:  int sum;
3:  for (int j = 0; j < myarray.length; j++) {
4:      for (int i = 0; j < myarray[0].length; i++) {
5:          myarray[i][j] = i + j;
6:          int sum = i + j;
7:      }
8:  }
9:  System.out.println(sum);
```

Bug 1:

Bug 2:

Bug 3:

Bug 4:

Bug 5:

Method Madness

3. (9 points) Consider the following program:

```
public class Madness {
    public static int[] y = { 7, 3, 2 };
    public static int   x = 2;

    public static void main(String[] args) {
        int x = Integer.parseInt(args[0]);
        x = mysterymethod(x, y);
        System.out.println(x);
        for (int i = 0; i < y.length; i++) {
            System.out.println(y[i]);
        }
    }

    public static int mysterymethod(int y, int[] z) {
        for (int i = 0; i < z.length; i++)
            z[i] += x;
        return x - y;
    }
}
```

(a) What will "`java Madness 7`" print out?

(b) What will "`java Madness 3`" print out?

(c) What will "`java Madness 2`" print out?

Why

4. (12 points) Consider the following class, then answer the questions on the following page:

```
public class Why {
    private String question;
    private String answer;

    public Why() { answer = "Because."; }

    public Why(String q) {
        question = q;
        answer   = "Why not?";
    }

    public Why(String q, String a) {
        question = q;
        answer = a;
    }

    public String question() {
        String q = "Why";
        if (question.length() > 0)
            q = q + " " + question;
        q = q + "?";
        return q;
    }

    public String answer()   {
        return answer;
    }

    public String toString() {
        return question() + "\n" + answer() + "\n";
    }
}
```

What will each of the following code snippets print? If the code will cause a runtime error, write `Exception`.

(a)
```
Why w = new Why();
System.out.println(w);
```

(b)
```
Why w = new Why("Why did the chicken cross the road?");
System.out.println(w);
```

(c)
```
Why w = new Why("Why did the chicken cross the road?", "To get to the other side.");
System.out.println(w);
```

(d)
```
Why w = new Why(null, "");
System.out.println(w);
```

(e)
```
Why w = new Why("", null);
System.out.println(w);
```

(f)
```
Why w = new Why(null, null);
System.out.println(w);
```

TOY

5. (13 points) Consider what happens when the following TOY program is executed by pressing RUN with the program counter set to 10:

```
01: 0001   ( 0000 0000 0000 0001 )
02: 0010   ( 0000 0000 0001 0000 )
03: 0012   ( 0000 0000 0001 0010 )
10: 8101   R[1] <- Mem[01]
11: C014   if (R[0] == 0) pc <- 14
12: 1331   R[3] <- R[3] + R[1]
13: 9302   Mem[02] <- R[3]
14: 8202   R[2] <- Mem[02]
15: 2321   R[3] <- R[2] - R[1]
16: C320   if (R[3] == 0) pc <- 20
17: 91FF   mem[FF] <- R[1]
18: 1111   R[1] <- R[1] + R[1]
19: C014   if (R[0] == 0) pc <- 14
20: 0000   halt
```

(a) Describe, in 20 words or less, what the above program does:

(b) Describe, in 20 words or less, what the program would do if the instruction at memory address 11 were replaced with the following:

```
11: 8302   R[3] <- Mem[02]
```

Whiteboarding

6. (20 points) The list of names of students waiting for help during office hours is a good example of a ring buffer. Students add their names to the end of the list, and TAs help students from the front of the list. When the list reaches the end of the board, it wraps around. But there is one wrinkle: students only get to have their name on the board at most once. Consider the following class to implement this behavior, then answer the two questions on the following pages:

```
public class Whiteboarding {
    private String[] rb; // the array backing the ring buffer
    private int first;   // index of the first valid element in the buffer
    private int last;    // index one past the last valid element in the buffer
    private int size;    // the number of elements currently in the buffer

    private Whiteboarding() {} // forbid the default constructor

    // Create a whiteboard that can hold capacity names
    public Whiteboarding(int capacity) {
      if (capacity <= 0) throw new RuntimeException("Invalid capacity.");
      rb = new String[capacity];
    }

    // return the number of names currently on the board
    public int size() { return size; }

    // return whether the board is empty/full
    public boolean isEmpty() { return size == 0; }
    public boolean isFull()  { return size == rb.length; }

    // Add name to the end of the queue if it is not
    // already in the queue, and the queue is not full
    public void enqueue(String name) { /* ... */ }

    // Delete and return the name from the front of the queue, or
    // return null if the queue is empty
    public String dequeue() { /* ... */ }

    // Return the name from the front of the queue, or null if the queue
    // is empty, but do not delete the element
    public String peek() { /* ... */ }
}
```

(a) Implement the `enqueue()` method so that it behaves according to its comment. Make sure the values of all instance variables are consistent with their comments when `enqueue()` returns. (Hint: recall that if `s` and `t` are strings, then `s.compareTo(t)` will return `0` if the two strings are the same.)

```
public void enqueue(String name) {

}
```

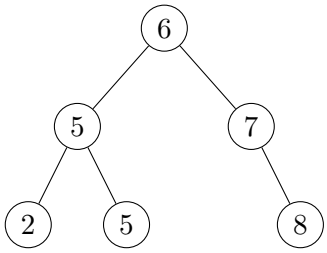
(b) What will the contents of the `rb` array be after the following sequence of statements? Indicate which entries in the array correspond to `first` and `last`

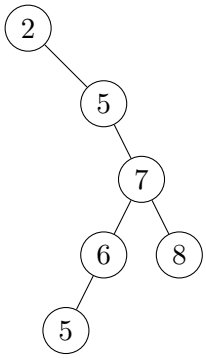
```
Whiteboarding w = new Whiteboarding(5);
w.enqueue("Dan");
w.enqueue("Jeff");
w.enqueue("Catherine");
w.enqueue("Dan");
w.enqueue("Rupi");
w.enqueue("Kathryn");
w.enqueue("Nate");
w.dequeue();
w.peek();
w.dequeue();
w.peek();
w.enqueue("Kathryn");
w.enqueue("Jeff");
w.enqueue("Nate");
```

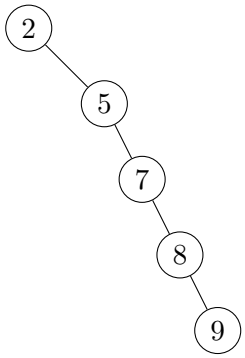
Your answer:

Binary Search Trees

7. (30 points) A binary search tree (BST) is a linked data structure, where each node contains a value and pointers to two other nodes, `left` and `right`. These nodes are referred to as its *children*. Just as the `next` node of a linked list is itself the start of a linked list, the `left` and `right` children of a BST node are themselves BSTs, called the left and right *sub-trees*. What makes BSTs interesting is an additional requirement that the every value in the left sub-tree of a node `n` must be less than the value stored in `n`, and every value stored in the right sub-tree of `n` must be greater than or equal to the value stored in `n`. If the left or right sub-trees of `n` are empty, then the `left` and/or `right` pointers will be `null`. Here are some examples of BSTs:







Searching for a value `v` in a BST `n` is very efficient using a simple recursive method. If `n.value == v` you're done. Otherwise, if `v < n.value` search the left sub-tree. Otherwise search the right sub-tree. If `n == null` then the tree doesn't contain `v`. Values can be added using a similar recursive method. A value `v` should be added to the left or right sub-tree depending on whether it is less than `n.value` or not. If that sub-tree is `null`, a new node containing the value `v` should be appended there. The nice thing about BSTs is that, if your values are well distributed, the size of the left and right sub-tree will be about equal and searching and insertion will be much faster than in a linked list. (More complex data structures, such as red-black trees guarantee this property.)

Consider the following `BST` class on the following page, then write the `insert()` and `printReverseSorted()` methods in the spaces provided.

```
public interface Comparable {
    // return 0 if this equal other
    //       -1 if this is less than other
    //        1 if this is greater than other
    int compareTo(Object other);
}

public class BST {
    private Comparable value;
    private BST left;
    private BST right;

    public BST(Comparable v) { value = v; }

    // return true if this tree contains v
    // or false if it doesn't
    public boolean find(Comparable v) {
        /* ... */
        int cmp = v.compareTo(value);
        if (cmp == 0) return true;
        else if (cmp < 0) {
            if (left != null) return left.find(v);
            else return false;
        } else {
            if (right != null) return right.find(v);
            else return false;
        }
    }

    // insert v into this tree
    public void insert(Comparable v) {
        /* ... */
    }

    // print the value of this node
    public void printValue() { System.out.println(value); }

    // print the tree contents in sorted order, one value per line
    public void printSorted() {
        /* ... */
    }

    // print the tree contents in reverse sorted order, one value per line
    public void printReverseSorted() {
        /* ... */
    }
}
```

(a) Implement the recursive `insert()` method as described in the problem description.

```
public void insert(Comparable v) {

}
```

(b) Implement the recursive `printReverseSorted()` method to print all the values in the tree in reversesorted order, as described in the problem description.

```
public void printReverseSorted() {

}
```

Postscript (extra paper)

Postscript (extra paper)