You have approximately two hours to complete this test. You may consult any books, notes, or other paper-based inanimate objects available to you. To avoid confusion, read the problems carefully. If you find it hard to understand a problem, ask us to explain it. If you have a question during the test, please come to the front or the side of the room to ask it.

You should assume that access to a key of a hash table takes constant time, and access to an element of a binary search tree takes at worst \( \log N \) time, where \( N \) is the number of elements in the tree. Anywhere you are asked to write a method, you may provide helper methods unless otherwise directed.

This exam comprises 15% of the points on which your final grade will be based. Partial credit may be given for wrong answers. Your exam should contain five problems (numbered 0 through 4) on 13 pages; a supplementary handout will also be distributed at the exam. Please write your answers in the spaces provided in the test. In particular, we will not grade anything on the back of an exam page unless we are clearly told on the front of the page to look there.

Some students are taking this exam late. Please do not talk to them, mail them information, or post anything about the exam to news groups or discussion forums until after Tuesday.

Relax—this exam is not worth having heart failure about.
**Problem 0 (1 point)**
Put your login name on each page. Also make sure you have provided the information requested on the first page.

**Problem 1 (2 points)**
Consider a version of the SchemeList class provided in lab. Its single instance variable is myHead; it includes a ConsNode inner class whose two instance variables are myCar and myCdr.

Write a SchemeList method named makeCircular that does nothing if this list is empty, and links the last node in this list to its first node otherwise.

```
// Precondition: this list isn’t circular.
// Postcondition: the last node in this list points to the first;
// this list is otherwise unchanged.
public void makeCircular ( ) {
```
Problem 2 (4 points)

Given below is an outline of a version of the BinarySearchTree class. An extra instance variable named myInfo has been added to the TreeNode class, to allow storage of information that would simplify certain operations on the tree.

```java
public class BinarySearchTree {
    private TreeNode myRoot;
    public BinarySearchTree ( ) {
        myRoot = null;
    }
    public boolean contains (Comparable obj) ...
    public void add (Comparable obj) ...
    public void remove (Comparable obj) ...
    private class TreeNode {
        public Comparable myItem;
        //public Object myInfo;
        public TreeNode myLeft, myRight;
        public TreeNode (Comparable obj) {
            myItem = obj;
            myInfo = null;
            myLeft = myRight = null;
        }
        public TreeNode (Comparable obj, TreeNode left, TreeNode right) {
            myItem = obj;
            myInfo = null;
            myLeft = left; myRight = right;
        }
    }
}
```

A use for the myInfo variable mentioned in one of the lab activities is to store a reference to the inorder successor of a node. An example appears in the diagram below.
**Part a**

Suppose we wish to define a class ThreadedBST that uses the myInfo variables as just described, by inheritance from BinarySearchTree:

```java
public class ThreadedBST extends BinarySearchTree ...
```

Indicate which lines in the BinarySearchTree class outline would need to be changed and what changes would be necessary in order to do this, while exposing as little as possible of the BinarySearchTree class.

**Part b**

Which of the following BinarySearchTree methods should one override in ThreadedBST in order to provide or take advantage of the threads?

- `add`
- `contains`
- `remove`

Briefly explain your answer(s).
Problem 3 (6 points)
The closest common ancestor of two amoebas a1 and a2 is the closest amoeba—the 
furthest from the root—that's an ancestor of both a1 and a2. (The ancestors of a 
given amoeba are itself, its parent, its grandparent, and so on.) In the tree below, 
mom/dad is the closest common ancestor of Wilma and Homer, Marge is the closest 
common ancestor of herself and Hilary, and me is the closest ancestor of Lisa and 
Homer.

We’ll assume for this problem that no two amoebas in the family have the same 
name.

Part a
An AmoebaFamily method to find the name of the closest ancestor of two named 
amoebas might be structured as follows:

```java
public String nameOfClosestCommonAncestor (String name1, String name2) {
    // First find the two relevant Amoeba objects, using a search method
    // as in one of the lab exercises.
    Amoeba a1 = search (name1);
    Amoeba a2 = search (name2);
    // Identify their closest common ancestor.
    Amoeba cca = closestCommonAncestor (a1, a2);
    // Return that ancestor's name.
    return cca.myName;
}
```

private static Amoeba closestCommonAncestor (Amoeba a1, Amoeba a2) ...

On the next page, provide the code for the closestCommonAncestor method. Your 
solution will be graded on efficiency: in particular, if your solution examines all the 
nodes in the tree when that’s not necessary, you will earn no credit on this part.

The instance variables in the Amoeba class are

```java
public String myName;       // amoeba's name
public Amoeba myParent;     // amoeba's parent; root's parent is null
public Vector myChildren;   // amoeba's children
```

The only methods in the Amoeba class are constructors.
Your answer to problem 3, part a

private static Amoeba closestCommonAncestor (Amoeba a1, Amoeba a2) {

Part b

Complete the following sentence. Then explain your answer briefly, making clear the meaning of whatever variables your running time estimate depends on. No matter how slowly your answer to part a runs (assuming it works), you can earn full credit on this part if you analyze it correctly.

The closestCommonAncestor method supplied as an answer to part a runs in time proportional to

________________________________________________________________

in the worst case.

Explanation:
Problem 4 (17 points)

Background

All parts of this problem involve a language for programming a robot. Commands in the robot language are typed one per line. A command sequence consists of 0 or more step, turnright, and repeat commands, followed by the command end (on a line by itself). The step, turnright, and repeat commands work as follows:

- step takes no arguments. It commands the robot to take one step in whatever direction it’s facing.
- turnright takes no arguments. It commands the robot to face 90 degrees to the right of the direction it’s currently facing.
- repeat takes a single nonnegative integer argument, and is followed by its loop body, a nested command sequence. It commands the robot to repeat the loop body the specified number of times. For example, the two segments below are equivalent:

```
repeat 3
  step
  turnright
end
```

```
step
  turnright
step
  turnright
step
  turnright
```

A program is a command sequence.

Here are some example programs, with loop bodies indented for clarity. The first example is an empty program. The fourth example has a nested loop with an empty body.

```
end
step
  turnright
step
  turnright
end
```

```
turnright
step
  repeat 4
    step
    turnright
end
turnright
step
  turnright
```

```
turnright
step
  repeat 4
    step
    turnright
end
end
turnright
step
end
```

(Background continues on the next page.)
Background, continued

The Program class outlined below represents a robot program. The Command class represents an individual command. The CommandSource class uses the InputSource class from project 2 (InputSource code is supplied in the supplementary handout).

Program.java

```java
public class Program {
    // The top-level command sequence.
    private Vector topLevel;
    // Constructor.
    public Program ( ) {
        ... 
    }
    // Other methods would go here.
}
```

CommandSource.java

```java
public class CommandSource {
    private InputSource lines;
    // InputSource is as in project 2.
    public CommandSource ( ) {
        lines = new InputSource ( );
    }
    // Return the next command in the input.
    // Precondition: we haven't reached
    // the end of the input.
    public Command next ( ) {
        String[ ] words = lines.nextLine();
        return new Command (words);
    }
}
```

Command.java

```java
public class Command {
    private String myName;
    private int myArg;
    private Vector myLoopBody;
    public Command (String [ ] cmdWords) {
        if (!cmdWords[0].equals ("repeat")) {
            myName = cmdWords[0];
            myArg = 0;
            myLoopBody = null;
        } else {
            myName = "repeat";
            try {
            myArg = Integer.parseInt(cmdWords[1]);
        } catch (NumberFormatException e) {
        }
        myLoopBody = new Vector ( );
    }
    public String name ( ) {
        return myName;
    }
    public int arg ( ) {
        return myArg;
    }
    public Vector loopBody ( ) {
        return myLoopBody;
    }
}
```

Robot programs are stored as structures similar to amoeba trees. The topLevel vector contains references to commands. Each repeat command contains a reference to a vector in which the commands of the loop body are stored; loop bodies may themselves contain repeat commands, and so on. The end commands that signal the end of command sequences serve merely to simplify input and are not stored internally.

(Background continues on the next page.)
Background, continued

Given below are box-and-pointer diagrams for some example robot programs.

turnright
step
turnright
step
turnright
step
turnright
step
turnright
step
turnright
step
turnright
step

repeat 4
step
turnright
end
turnright
step
end

repeat 4
step
turnright
repeat 3
end
turnright
step
end

repeat 3
(0-element vector)
Part a
Complete the fill method called by the Program constructor. Don’t worry about illegal input.

```java
public Program () {
    CommandSource cmds = new CommandSource ( );
    topLevel = new Vector ( );
    fill (topLevel, cmds);
}
```

// Fill the given vector with successive commands from the command source,
// as just described.
```java
private void fill (Vector v, CommandSource cmds) {
```

Your login name: cs61b-________
Part b

It is possible to determine how many step and turnright commands are executed in a given robot program. For example, the program

```java
turnright
step
repeat 4
  step
  turnright
  repeat 3
    turnright
  end
end
```

executes 24 step and turnright commands in all.

Complete the Program moveCount method below. It should return the total number of step and turnright commands that will be executed when the robot program is run. Assume that the internal representation of the robot program is correct.

```java
public int moveCount() {
    // Your code here
}
```
Part c

Given below is the code for the Command constructor. It doesn’t check its argument for errors. Rewrite the constructor to throw an `IllegalInputException` with no message if the format of its argument line is incorrect in any way.

```java
public Command (String [ ] cmdWords) {
    if (!cmdWords[0].equals("repeat")) {
        myName = cmdWords[0];
        myArg = 0;
        myLoopBody = null;
    } else {
        myName = "repeat";
        try {
            myArg = Integer.parseInt(cmdWords[1]);
        } catch (NumberFormatException e) {
        }
        myLoopBody = new Vector ( );
    }
}
```