

NOTE: The exam required knowledge of your TA's name, lab time, and login. It was open book and open notes. ALSO, THIS WAS LAB BASED 61B (hence many lab related questions).

Problem 0 (1 point)

Put your login name on each page. Also make sure you have the provided 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 the first node otherwise.

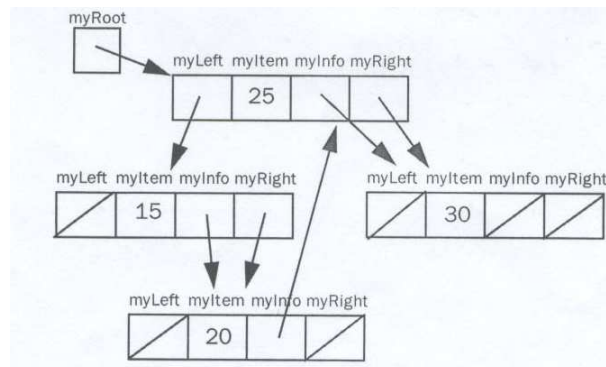
```
// Precondition: This list isn't circular.
// Postcondition: The last node in this list points to the first;
//               this list is otherwise unchanged.
public void makeCircular(){
```

Problem 2 (4 points)

Given below is an outline of 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.

```
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`:

```
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 `ThreadedTree` in order to provide or take advantage of the threads?

Add

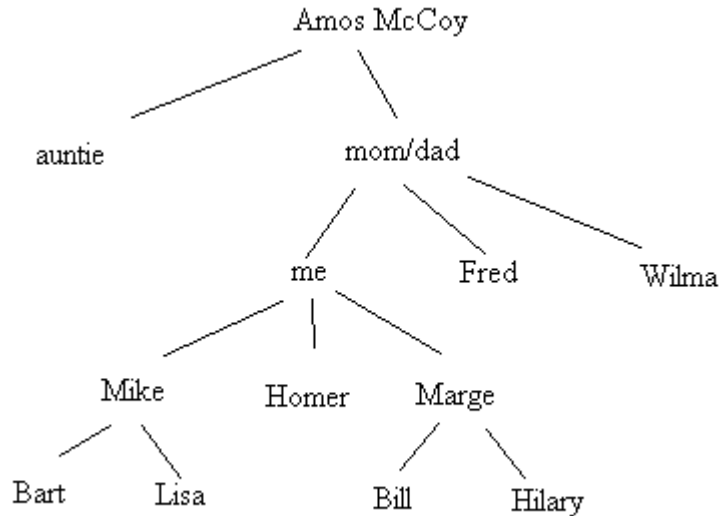
Contains

Remove

Briefly explain your answer(s).

Problem 3 (6 points)

The *closest common ancestor* of two amoebas a_1 and a_2 is the closest amoeba – the furthest from the root – that’s an ancestor of both a_1 and a_2 . (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:

```
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 a1)...
```

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 it’s not necessary, you will earn no credit on this part.

The instance variables in the `Amoeba` class are

```
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 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
                 step   turnright
                 turnright step
                 end     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          turnright          turnright          turnright
             Step              step              step
             Turnright         repeat 4         repeat 4
             Step              step              step
             End                turnright        turnright
                                 End              repeat 3
                                 Turnright        end
                                 step              End
                                 end              turnright
                                           turnright
                                           step
                                           end
```

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

Program.java

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

CommandSource.java

```
Public class CommandSource{
    Private InputSource lines;
    // InputSource 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

```
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 = "repear";
            try {
                myArg =
                    Integer.parseInt(cmdWords[1]);
            } catch (NumberFormatException e){
            }
            myLoopBody = new Vector();
        }
    }

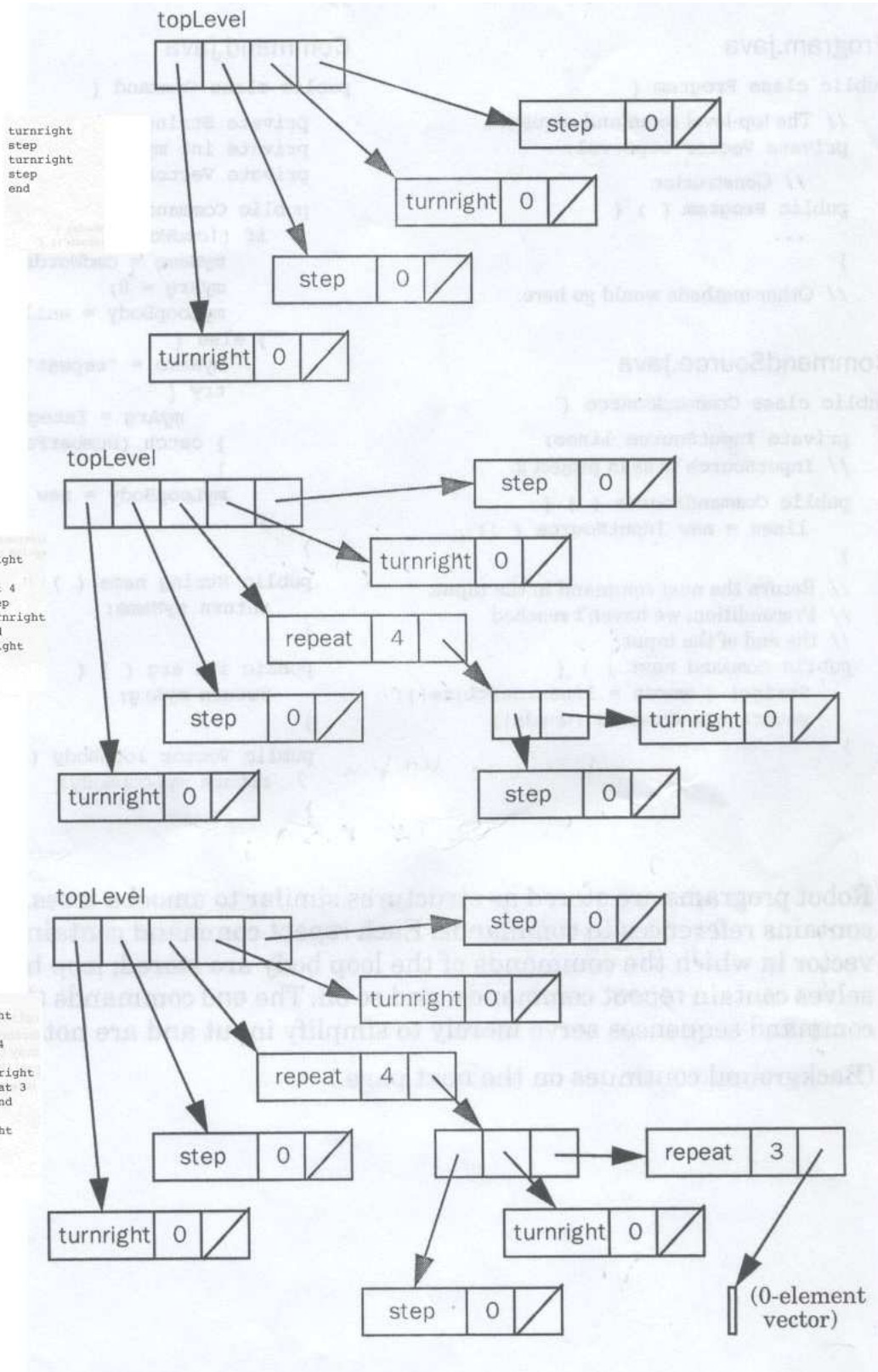
    public String name () {
        return myName;
    }

    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.

The following are box-and-pointer diagrams for some example robot programs.



Part a

Complete the `fill` method called by the `Program` constructor. Don't worry about illegal input.

```
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.

Private void fill (Vector v, CommandSource cmds) {
```

Part b

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

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

4 step and 16 turnright commands contributed by the outer loop

3 turnright commands per outer loop execution

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.

```
Public int moveCount(){
```

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 `IllegalArgumentException` with no message if the format of its argument line is incorrect in any way.

```
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();
    }
}
```