Comp 212 - Intermediate Programming EXAM #2 March 29, 2006
Rice University - Instructors: Cox & Nguyen NAME: ______________________

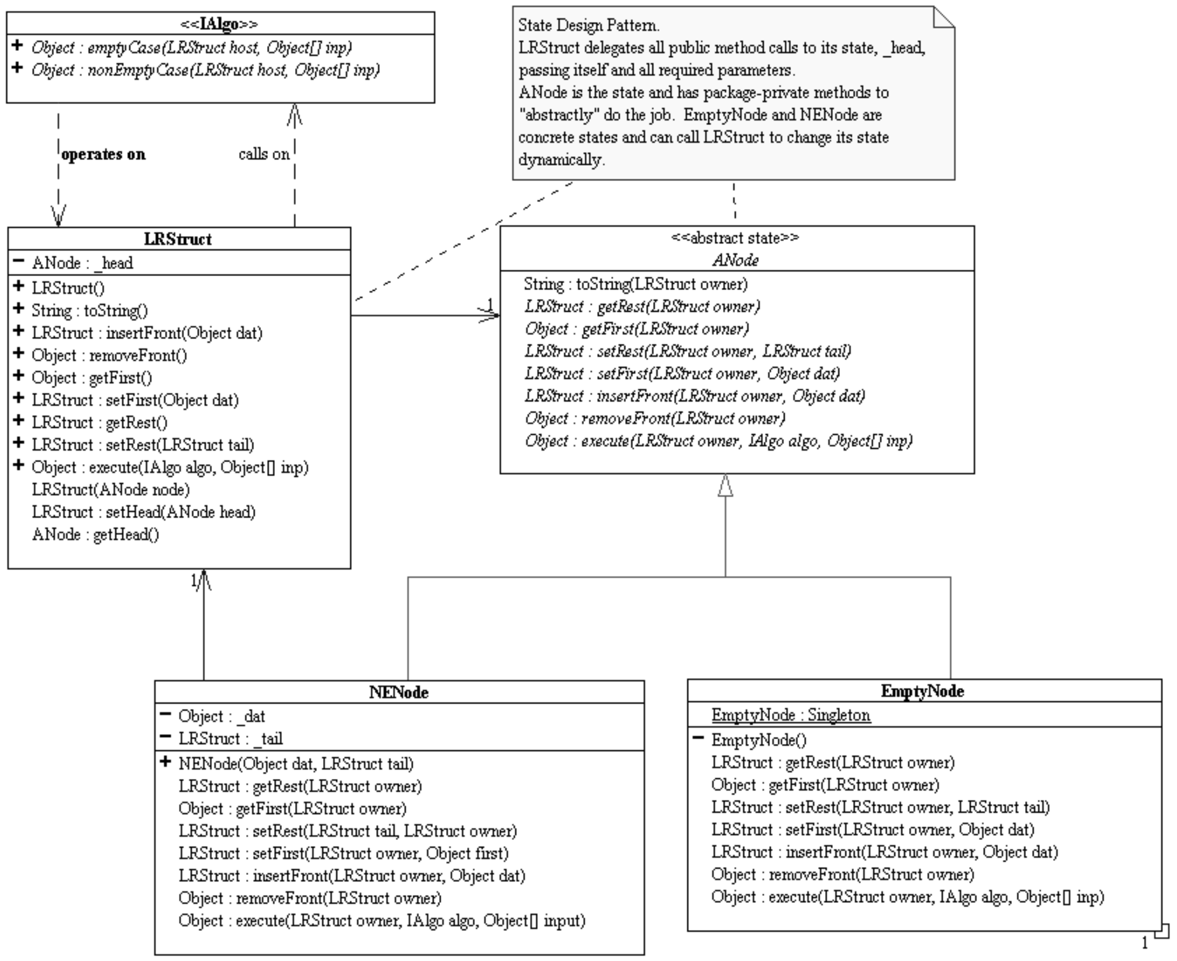
Instructions

1. This exam is conducted under the Rice Honor Code. It is an open-book exam. You may use lecture notes, lab notes, homework assignments, solutions provided to you by the teaching staff, your own code, code written in cooperation with others before the exam.
2. Fill in your name on every page of the exam.
3. If you forget the name of a Java class or method, make up a name for it and write a *brief* explanation in the margin.
4. You are expected to know the syntax of defining a class with appropriate fields, methods, and inheritance hierarchy. You will not be penalized on trivial syntax errors, such as missing curly braces, missing semi-colons, etc, but do try to write Java code as syntactically correct as possible. We are more interested in your ability to show us that you understand the concepts than your memorization of syntax!
5. Write your code in the most object-oriented way possible, that is, with the fewest number of control statements and no checking of the states and class types of the objects involved.
6. In all of the questions, feel free to write additional helper methods or visitors to get the job done.
7. For each algorithm you are asked to write, 90% of the grade will be for correctness, and 10% will be for efficiency and code clarity.
8. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

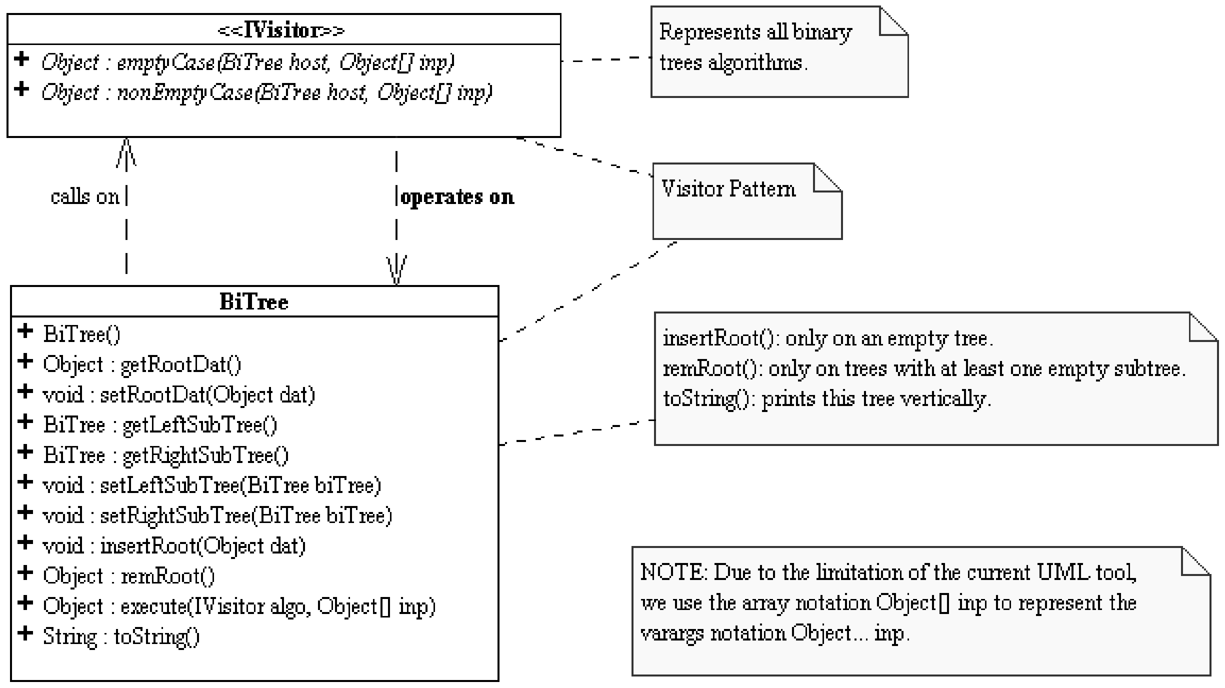
1a) 10 pts	1b) 10 pts	1c) 10 pts	2) 20 pts	3a) 20 pts	3b) 5 pts	4) 25 pts	Total 100 pts

For your convenience, below is the UML class diagrams for LRStruct and BiTree studied in class. You are free to use these classes without explanation/implementation.



<<IAlgo>>
+ Object : emptyCase(LRStruct host, Object[] inp)
+ Object : nonEmptyCase(LRStruct host, Object[] inp)
operates on
calls on
LRStruct
− ANode : _head
+ LRStruct()
+ String : toString()
+ LRStruct : insertFront(Object dat)
+ Object : removeFront()
+ Object : getFirst()
+ LRStruct : setFirst(Object dat)
+ LRStruct : getRest()
+ LRStruct : setRest(LRStruct tail)
+ Object : execute(IAlgo algo, Object[] inp)
LRStruct(ANode node)
LRStruct : setHead(ANode head)
ANode : getHead()
1
<<abstract state>>
ANode
String : toString(LRStruct owner)
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct owner, LRStruct tail)
LRStruct : setFirst(LRStruct owner, Object dat)
LRStruct : insertFront(LRStruct owner, Object dat)
Object : removeFront(LRStruct owner)
Object : execute(LRStruct owner, IAlgo algo, Object[] inp)
1
NENode
− Object : _dat
− LRStruct : _tail
+ NENode(Object dat, LRStruct tail)
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct tail, LRStruct owner)
LRStruct : setFirst(LRStruct owner, Object first)
LRStruct : insertFront(LRStruct owner, Object dat)
Object : removeFront(LRStruct owner)
Object : execute(LRStruct owner, IAlgo algo, Object[] input)
EmptyNode
EmptyNode : Singleton
− EmptyNode()
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct owner, LRStruct tail)
LRStruct : setFirst(LRStruct owner, Object dat)
LRStruct : insertFront(LRStruct owner, Object dat)
Object : removeFront(LRStruct owner)
Object : execute(LRStruct owner, IAlgo algo, Object[] inp)
1

State Design Pattern.
LRStruct delegates all public method calls to its state, _head, passing itself and all required parameters.
ANode is the state and has package-private methods to "abstractly" do the job. EmptyNode and NENode are concrete states and can call LRStruct to change its state dynamically.


<<IVisitor>>
+ Object : emptyCase(BiTree host, Object[] inp)
+ Object : nonEmptyCase(BiTree host, Object[] inp)
Represents all binary trees algorithms.
calls on
operates on
Visitor Pattern
BiTree
+ BiTree()
+ Object : getRootDat()
+ void : setRootDat(Object dat)
+ BiTree : getLeftSubTree()
+ BiTree : getRightSubTree()
+ void : setLeftSubTree(BiTree biTree)
+ void : setRightSubTree(BiTree biTree)
+ void : insertRoot(Object dat)
+ Object : remRoot()
+ Object : execute(IVisitor algo, Object[] inp)
+ String : toString()
insertRoot(): only on an empty tree.
remRoot(): only on trees with at least one empty subtree.
toString(): prints this tree vertically.
NOTE: Due to the limitation of the current UML tool, we use the array notation Object[] inp to represent the varargs notation Object... inp.

1. **Definition of Palindrome**: According to Webster.com, a palindrome is a word, verse, or sentence (as "Able was I ere I saw Elba") or a number (as 1881) that reads the same backward or forward.

   a. Write a method called `isPalindrome` that takes in as input parameter an array of char and returns true when the array of char represents a palindrome, false otherwise. Just assume the method is part of some utility class. Feel free to write any additional methods if needed.

```
public boolean isPalindrome(char[] w) {
// students to complete






















}
```

b. Write a visitor for `LRStruct` called `IsPalin` that returns `Boolean.TRUE` when the host `LRStruct` represents a palindrome, `Boolean.FALSE` otherwise. This visitor must compute the reverse of the host and compare the host against its reverse for equality of each of the elements in the `LRStruct`. Feel free to write additional helper visitors if needed.

```
public class IsPalin implements IAlgo {
    public final static IsPalin Singleton = new Contain();
    private IPalin() {
    }

    public Object emptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE







    }

    public Object nonEmptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE















    }
}
```

c. Observe the following:

   i. An empty `LRStruct` is a palindrome.
   ii. A one-element `LRStruct` is a palindrome.
   iii. An `LRStruct` with more than one element is a palindrome if and only if its first and last elements are equal and the sub `LRStruct` with the first and last elements removed is a palindrome.

Translate the above observation into a visitor for `LRStruct` called `IsPalin2` that returns `Boolean.TRUE` when the host `LRStruct` represents a palindrome, `Boolean.FALSE` otherwise. You may use the visitor `RemLast` (discussed in lecture 14) that removes the last element in the host `LRStruct` without explanation. In this visitor, it is acceptable for the host `LRStruct` to mutate. Feel free to write additional helper visitors if needed.

```
public class IsPalin2 implements IAlgo {
    public final static IsPalin Singleton = new Contain();
    private IPalin() {
    }

    public Object emptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE




    }

    public Object nonEmptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE




    }
}
```

2. Consider a binary search tree, specifically, the implementation presented in lecture using class `BiTree`. Write a `BiTree` visitor named `BSTFindNext` that returns the `BiTree` object that is the next largest object relative to the visitor's input. The visitor's input may or may not be in the binary search tree. If there is no next larger element, return an empty `BiTree`.

Examples

Consider the following binary search tree containing Integer.

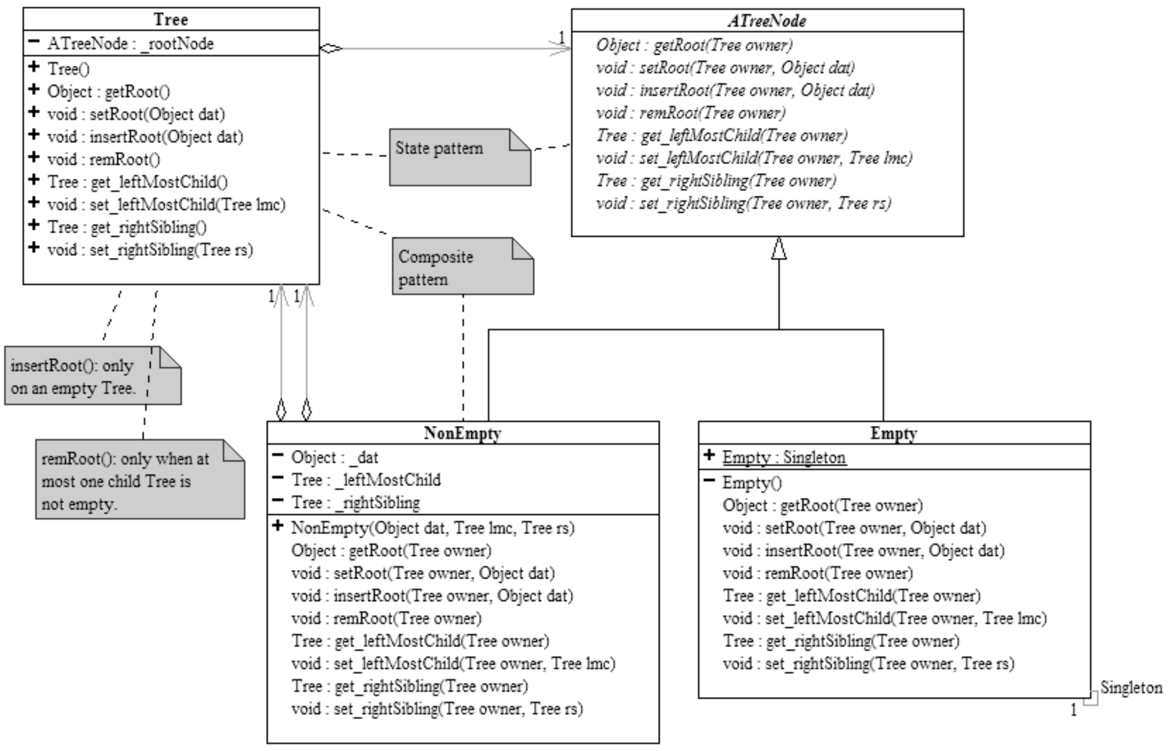
```
   57

  /  \

33    78

     /  \

   63    91

  /  \

59    71

     /

   70
```

Let `BiTree bst` be the above binary search tree and `bstfindnext` be an instance of `BSTFindNext`.

- `bst.execute(bstfindnext, 63)` would return the `BiTree` containing 70 at the root.
- `bst.execute(bstfindnext, 71)` would return the `BiTree` containing 78 at the root.
- `bst.execute(bstfindnext, 60)` would return the `BiTree` containing 63 at the root.
- `bst.execute(bstfindnext, 91)` would return an empty `BiTree`.

```
public class BSTFindNext implements IVisitor {
    private Comparator _order;
    /**
     * Used when the items in the tree are Comparable objects.
     */
    public BSTFindNext () {
        _order = new Comparator() {
            public int compare(Object x, Object y) {
                return ((Comparable)x).compareTo(y);
            }
        };
    }

    /**
     * Used when the items are ordered according to a given Comparator.
     * @param order a total ordering on the items stored in the tree.
     */
    public BSTFindNext (Comparator order) {
        _order = order;
    }
    public Object emptyCase(BiTree host, Object... nu) {
        // STUDENT TO COMPLETE







    }


    public Object nonEmptyCase(BiTree host, Object... input) {
        // STUDENT TO COMPLETE
























    }
}
```

3. Below is the UML class diagram representing an n-ary tree (tree with an arbitrary number of subtrees).


Tree
− ATreeNode : _rootNode
+ Tree()
+ Object : getRoot()
+ void : setRoot(Object dat)
+ void : insertRoot(Object dat)
+ void : remRoot()
+ Tree : get_leftMostChild()
+ void : set_leftMostChild(Tree lmc)
+ Tree : get_rightSibling()
+ void : set_rightSibling(Tree rs)
ATreeNode
Object : getRoot(Tree owner)
void : setRoot(Tree owner, Object dat)
void : insertRoot(Tree owner, Object dat)
void : remRoot(Tree owner)
Tree : get_leftMostChild(Tree owner)
void : set_leftMostChild(Tree owner, Tree lmc)
Tree : get_rightSibling(Tree owner)
void : set_rightSibling(Tree owner, Tree rs)
State pattern
Composite pattern
insertRoot(): only on an empty Tree.
remRoot(): only when at most one child Tree is not empty.
1
1
1
NonEmpty
− Object : _dat
− Tree : _leftMostChild
− Tree : _rightSibling
+ NonEmpty(Object dat, Tree lmc, Tree rs)
Object : getRoot(Tree owner)
void : setRoot(Tree owner, Object dat)
void : insertRoot(Tree owner, Object dat)
void : remRoot(Tree owner)
Tree : get_leftMostChild(Tree owner)
void : set_leftMostChild(Tree owner, Tree lmc)
Tree : get_rightSibling(Tree owner)
void : set_rightSibling(Tree owner, Tree rs)
Empty
+ Empty : Singleton
− Empty()
Object : getRoot(Tree owner)
void : setRoot(Tree owner, Object dat)
void : insertRoot(Tree owner, Object dat)
void : remRoot(Tree owner)
Tree : get_leftMostChild(Tree owner)
void : set_leftMostChild(Tree owner, Tree lmc)
Tree : get_rightSibling(Tree owner)
void : set_rightSibling(Tree owner, Tree rs)
Singleton
1

- An n-ary tree can be empty or non-empty.
- An empty n-ary tree contains nothing.
- A non-empty n-ary tree contains a data element, an n-ary tree called the left most child tree and an n-ary tree called the right sibling tree.

Example: Using horizontal lines to denote right sibling links and (slanted) vertical lines to denote left-most child links, and the letter 'e' to denote an empty n-ary tree, the binary tree from problem 2 is represented in this model as

```
     57 -- e
    /
  33 -- 78 -- e
 /     /
e    63 -- 91 -- e
    /
  59 -- 71 -- e
 /     /
e    70 -- e
    /
   e
```

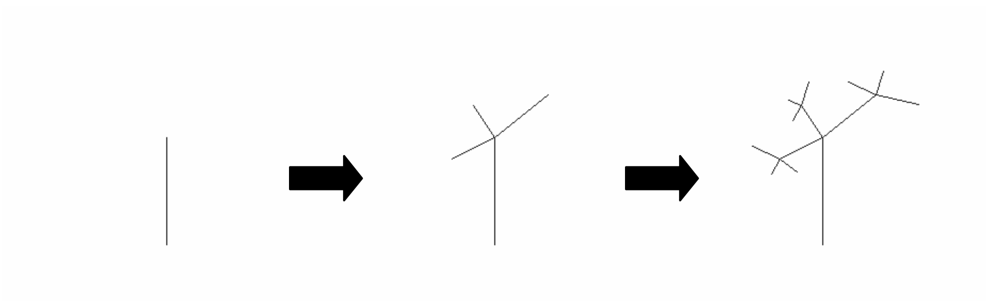
a. Write a `BiTree` visitor called `MakeTree` that returns an equivalent n-ary tree (`Tree`) of the `BiTree` host. Feel to write additional helper visitors if needed.

```
public class MakeTree implements IVisitor {
// Singleton Pattern code elided…

    public Object emptyCase(BiTree host, Object... nu) {
        // STUDENT TO COMPLETE


    }


    public Object nonEmptyCase(BiTree host, Object... input) {
        // STUDENT TO COMPLETE

    }
}
```

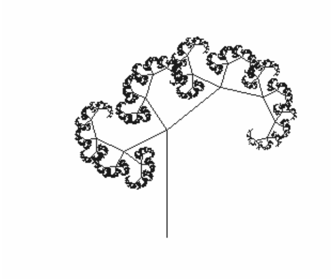
b. If you are to add the visitor pattern to the above design, what should the visitor interface be?

4. In this problem we will be exploring a new type of fractal: Tree Fractals.

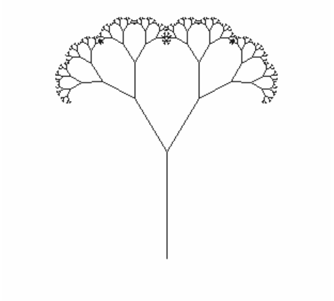
A tree fractal starts as a single “line”. If we “grow” the line, it transforms into a “branch”, which is the original line, plus (in this case, 3) lines "sprouting" from its endpoint. These new lines are sized and are pointing relative to the original line. We see that clearly if we grow the tree fractal again:



The process simulates the way a plant grows. After 8 iterations, we see a distinctly organic shape forming:



Modifying the way a line grows into a branch changes the way the tree fractal looks:



Tree fractals, differentiated by their branching properties, can be used to describe the growth differences between broccoli and cauliflower and between various species of trees such as an elm vs. an oak.

Your task is to design an object model of a tree fractal. Your design should comply with the following requirements:

- There should be a top-level class called `TreeFractal` that represents a tree fractal of any complexity or growth level.
- `TreeFractal` should be able to perform at least two following behaviors:
  - Paint the fractal onto a supplied Graphics object and
  - Grow the fractal by mutation.
- Your model and comments should be complete enough to fully describe how the above behaviors are implemented. Giving examples of usage is generally very helpful in clarify your points.
- You should include discussions of what design patterns you used and why.
- The growth characteristics of the tree fractal should be dynamically changeable.
- All methods signatures need to be complete but the method bodies need only contain comments as to what they would be expected to do. Your comments should be complete and detailed enough to accurately

describe what is going on in that method. You do not, however, need to go to the detailed level of describing things such as affine transformations or traversals of prototype lists.