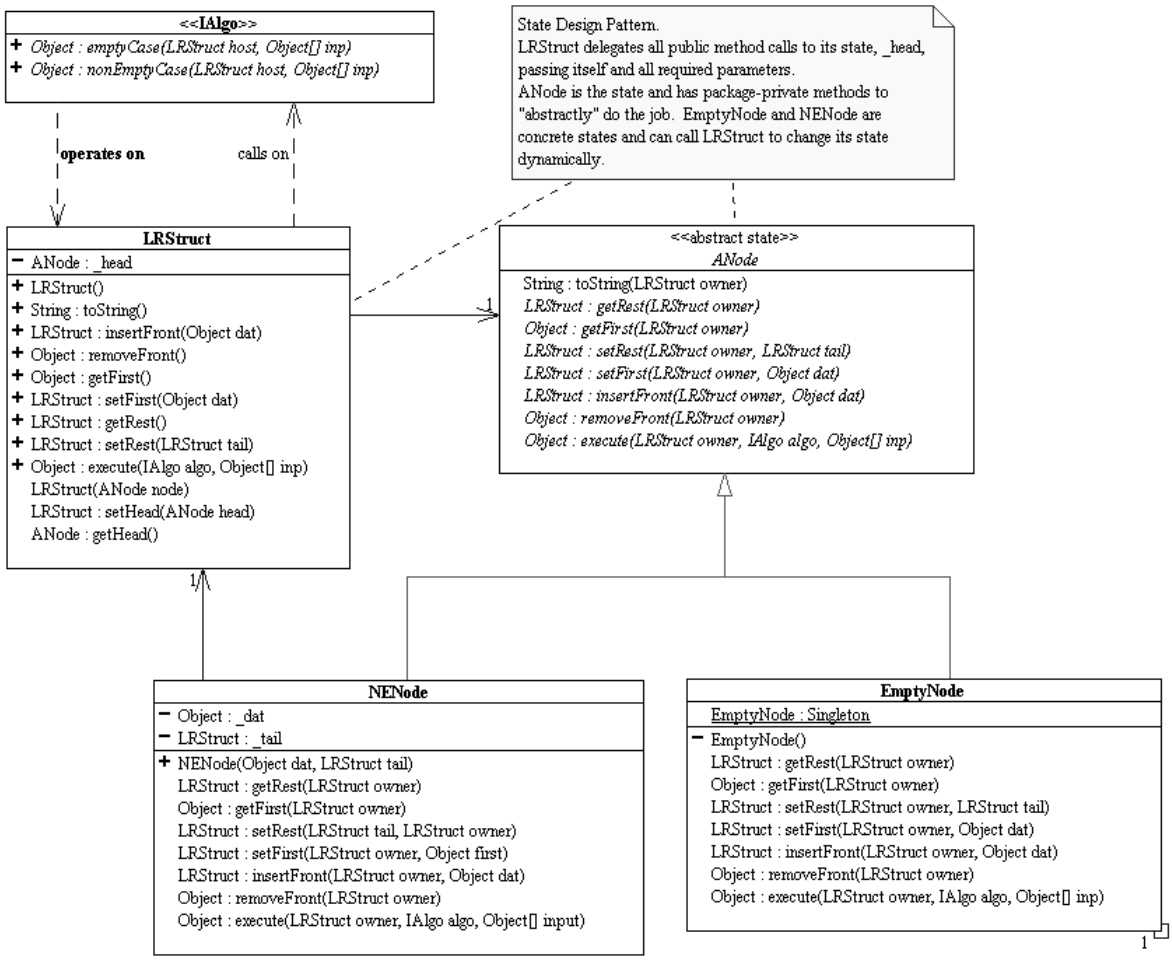
Instructions

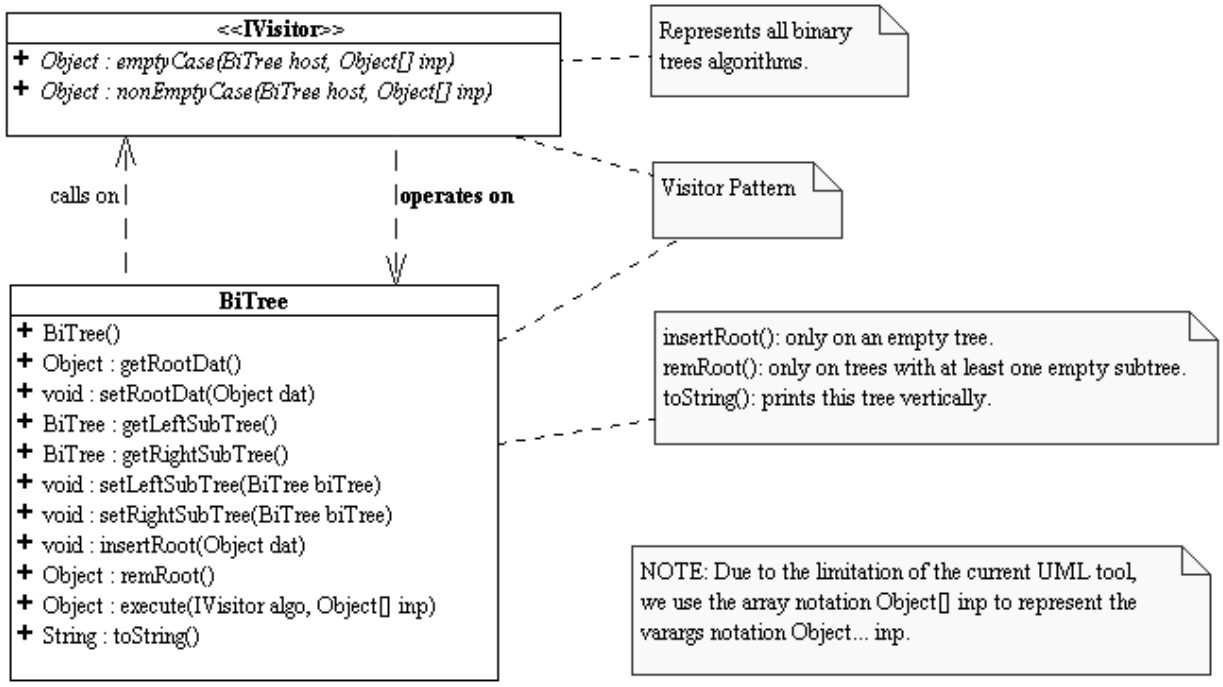
1. This exam is conducted under the Rice Honor Code. It is an open-book exam. You may use lecture notes, lab notes, homework assignments, solutions provided to you by the teaching staff, your own code, code written in cooperation with others before the exam.
2. Fill in your name on every page of the 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) 5 pts	1b) 5 pts	1c) 5 pts	2) 25 pts	3) 30 pts	4) 30 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.

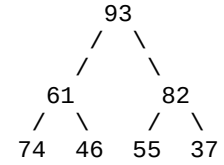



<<IVisitor>>
+ Object : emptyCase(BiTree host, Object[] inp)
+ Object : nonEmptyCase(BiTree host, Object[] inp)
Represents all binary trees algorithms.
calls on
operates on
Visitor Pattern
BiTree
+ BiTree()
+ Object : getRootDat()
+ void : setRootDat(Object dat)
+ BiTree : getLeftSubTree()
+ BiTree : 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. (15 pts Total) Consider the following program

```
import brs.*;
import genRac.*;

public class TreeTraversal {

    /**
     * Traverse a BiTree using a RAC to control the order of traversal.
     * @param root the BiTree to traverse
     * @param factory a factory for creating a RAC
     */
    public static void traverse(BiTree root, IRACFactory<BiTree> factory) {
        final IRAContainer<BiTree> rac = factory.makeRAC();

        /**
         * If root is non-empty, add it to rac.
         */
        root.execute(new IVisitor (){
            public Object emptyCase(BiTree host, Object... input) {
                return null;
            }

            public Object nonEmptyCase(BiTree host, Object... input) {
                rac.put(host);
                return null;
            }
        });

        System.out.print("Traversal:");

        while (!rac.isEmpty()) {
            BiTree bt = rac.get();

            System.out.print(" " + bt.getRootDat());

            /**
             * If bt's left child is non-empty, add it to rac.
             */
            bt.getLeftSubTree().execute(new IVisitor (){
                public Object emptyCase(BiTree host, Object... input) {
                    return null;
                }

                public Object nonEmptyCase(BiTree host, Object... input) {
                    rac.put(host);
                    return null;
                }
            });

            /**
             * If bt's right child is non-empty, add it to rac.
             */
            bt.getRightSubTree().execute(new IVisitor (){
                public Object emptyCase(BiTree host, Object... input) {
                    return null;
                }

                public Object nonEmptyCase(BiTree host, Object... input) {
                    rac.put(host);
                    return null;
                }
            });
        }

        System.out.println("");
    }
}
```

Your task is to write down the precise output of TreeTraversal.traverse() for each of the following cases. In all cases, the following binary tree of integers is passed to TreeTraversal.traverse().



a) (5 pts) What is the output if an LRSStackFactory<BiTree> is passed to TreeTraversal.traverse()?

b) (5 pts) What is the output if an LRSQueueFactory<BiTree> is passed to TreeTraversal.traverse().

c) (5 pts) What is the output if an PQComparatorRACFactory<BiTree> is passed to TreeTraversal.traverse(). Note: Remember that the RACs created by PQComparatorRACFactory<BiTree> return the largest item in the RAC first.

2. (25 pts) Given below is the code to delete a node from a binary search tree. This code works because the method remRoot() of BiTree only requires that at least one of the subtrees be empty. Rewrite the deletion algorithm for a binary search tree in the case where remRoot() requires that both subtrees are empty. Feel free to write any additional helper visitors.

Given code:

```
public class BSTDeleter implements IVisitor {
    private Comparator _order;

    public BSTDeleter() {
        _order = new Comparator() {
            public int compare(Object x, Object y) {
                return ((Comparable)x).compareTo(y);
            }
        };
    }

    public BSTDeleter (Comparator order) {
        _order = order;
    }

    public Object emptyCase(BiTree host, Object... nu) {
        return Boolean.FALSE;
    }

    public Object nonEmptyCase(final BiTree host, Object... input) {
        Object root = host.getRootDat();
        if (_order.compare(input[0], root) < 0) {
            return host.getLeftSubTree().execute (this, input[0]);
        }
        if (_order.compare(input[0], root) > 0) {
            return host.getRightSubTree().execute (this, input[0]);
        }
        return host.getLeftSubTree().execute(new IVisitor() {
            public Object emptyCase (BiTree h, Object notUsed) {
                host.remRoot();
                return Boolean.TRUE;
            }

            public Object nonEmptyCase (BiTree h, Object notUsed) {
                BiTree maxTree = (BiTree)h.execute(MaxTreeFinder.Singleton);
                host.setRootDat(maxTree.remRoot());
                return Boolean.TRUE;
            }
        });
    }
}

/**
 * Returns the subtree of the host with the max value in the root if the tree
 * is not empty, otherwise returns the host itself, assuming the tree satisfies
 * the BST property.
*/
public class MaxTreeFinder implements IVisitor {
    public static final MaxTreeFinder Singleton = new MaxTreeFinder ();
    private MaxTreeFinder ()  {
    }

    public Object emptyCase(BiTree host, Object... nu) {
        return host;
    }
```

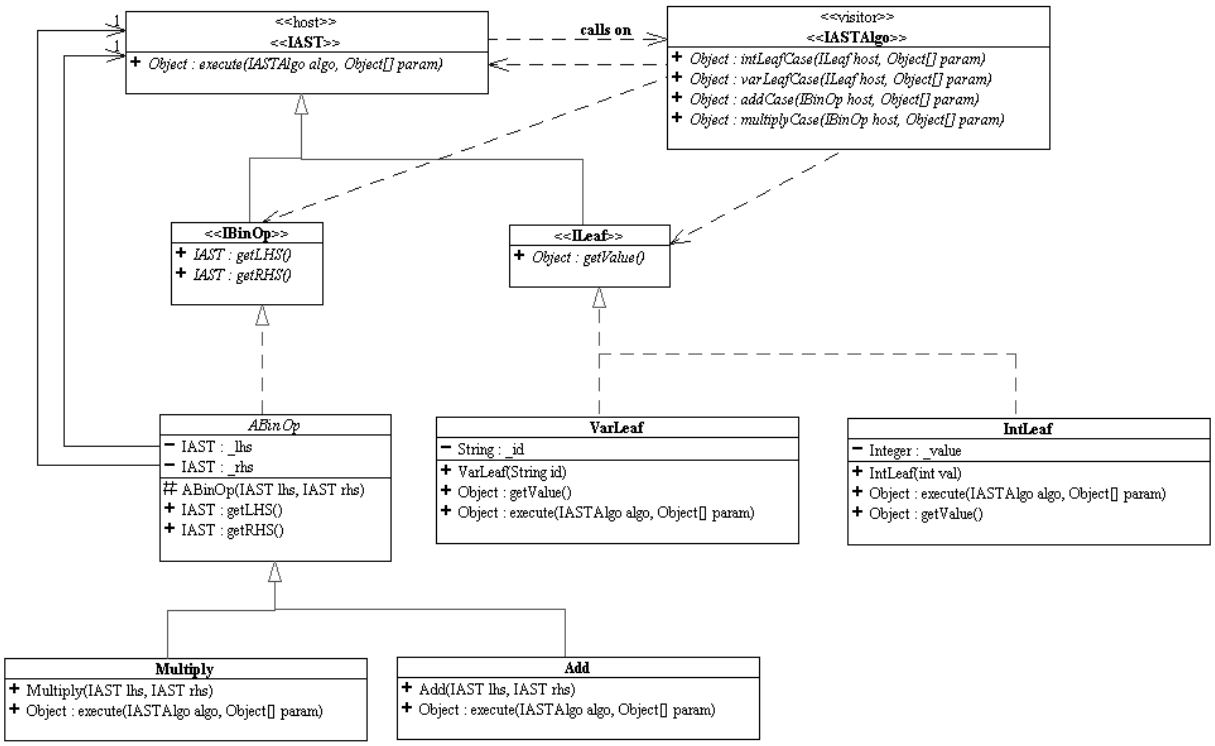
```
    public Object nonEmptyCase (BiTree host, Object... nu)  {
        return host.getRightSubTree().execute (new IVisitor () {
            public Object emptyCase (BiTree h, Object... hp) {
               return hp[0];
            }

            public Object nonEmptyCase (BiTree h, Object... hp) {
               return h.getRightSubTree ().execute (this, h);
            }
        }, host);
    }
}
```

Your code:

```
public class BSTDeleter2 implements IVisitor {
    private Comparator _order;

    public BSTDeleter2() {
        _order = new Comparator() {
            public int compare(Object x, Object y) {
                return ((Comparable)x).compareTo(y);
            }
        };
    }

    public BSTDeleter2 (Comparator order) {
        _order = order;
    }

    public Object emptyCase(BiTree host, Object... nu) {
        return Boolean.FALSE;
    }

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









    }
}
```

This page is left blank for code.

3. (30 pts) Below is the UML class diagram representing the abstract syntax tree (AST) object model for arithmetic expressions given in homework #4.



Write an IASTAlgo to simplify arithmetic expressions with constants into single constants within an AST. The visitor should return the simplified AST. Since the AST is immutable, the visitor should construct new AST objects as necessary to account for changes in the simplification process. Feel free to write any additional helper visitors.

Examples:

host AST	output AST
57	57
xyz	xyz

`  +`
`/ \`
`33 8`
	41

`  +`
`/ \`
`33 *`
`/ \`
`+ xyz`
`/ \`
`5 4`
	`  +`
	`/ \`
	`33 *`
	`/ \`
	`9 xyz`

```
public class Simplify implements IASTAlgo{
    public final static Simplify Singleton = new Simplify();
    private Simplify() {
    }

    public Object intLeafCase(ILeaf host, Object... nu) {
        // student to complete



    }

    public Object varLeafCase(ILeaf host, Object... nu) {
        // student to complete



    }

    public Object addCase(IBinOp host, Object... nu) {
        // student to complete









    }

    public Object multiplyCase(IBinOp host, Object... nu) {
        // student to complete









    }
}
```

This page is left blank for code.

4. (30 pts) Write an IAlgo visitor for LRStruct called Weave that mutates the host LRStruct and the input LRStruct such that the two lists become equivalent but do not share any NENode objects. In other words, they remain two distinct lists that contain the same data objects in the same order. The data objects in the two lists should be weaved together in alternation as illustrated by the following example. When you weave L1 = (5, 3, 9) together with L2 = (-7, 0, 8, -15), the result is both L1 and L2 *mutate* to become the list (5, -7, 3, 0, 9, 8, -15) but remain *distinct* and do *not* share any NENode objects. You may assume that the host list and the input list are distinct and do not share any NENode to start with. Note that the lists may have different lengths. Feel free to write any additional helper visitors.

```
public class Weave implements IAlgo {
    public static final Weave Singleton = new Weave();
    private Weave() {
    }
    /**
     * @param host empty
     * @param input[0] the other LRStruct.
     */
    public Object emptyCase(LRStruct host, Object... input) {
        // student to complete




    }
    /**
     * @param host not empty
     * @param input[0] the other LRStruct.
     */
    public Object nonEmptyCase(LRStruct host, Object... input) {
        // student to complete







    }
}
```

This page is left blank for code.