 NAME & ID#: ______________________________

Instructions

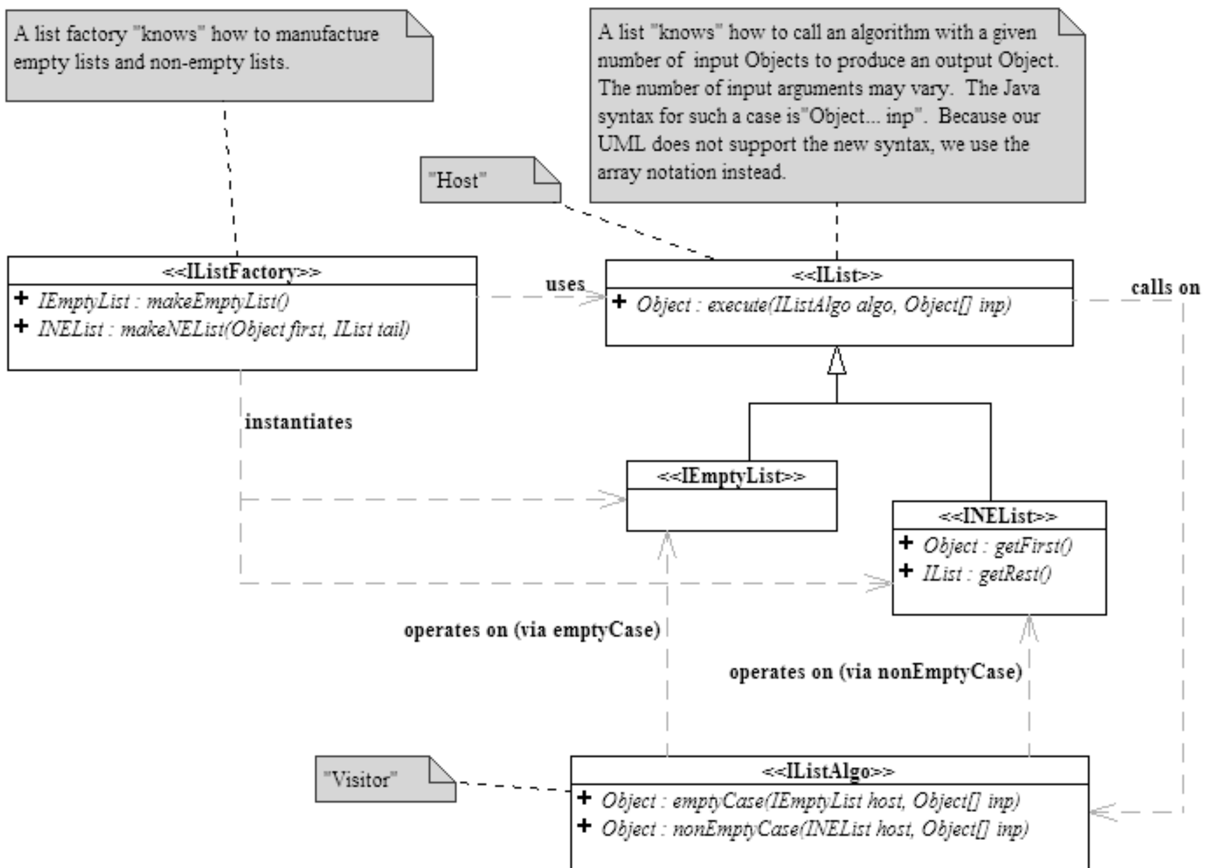
1. This is an open-notes, open-book, open-computer, open-Internet exam.
2. **You have 2.5 hours to complete this exam.**
3. Please make sure you have all 16 pages of this exam.
4. You will not be penalized on trivial syntax errors, such as a missing parenthesis. Multiple errors or errors that lead to ambiguous code will have points deducted, however.
5. In all of the questions, feel free to write additional helper methods to get the job done.
6. The emphasis is on correctness of the code, not efficiency or on simply generating the right result.
7. You are free to use any code that was given to you in the lectures and labs.
8. You do not have to write generic code in this exam.

Please write and sign the Rice Honor Pledge here:

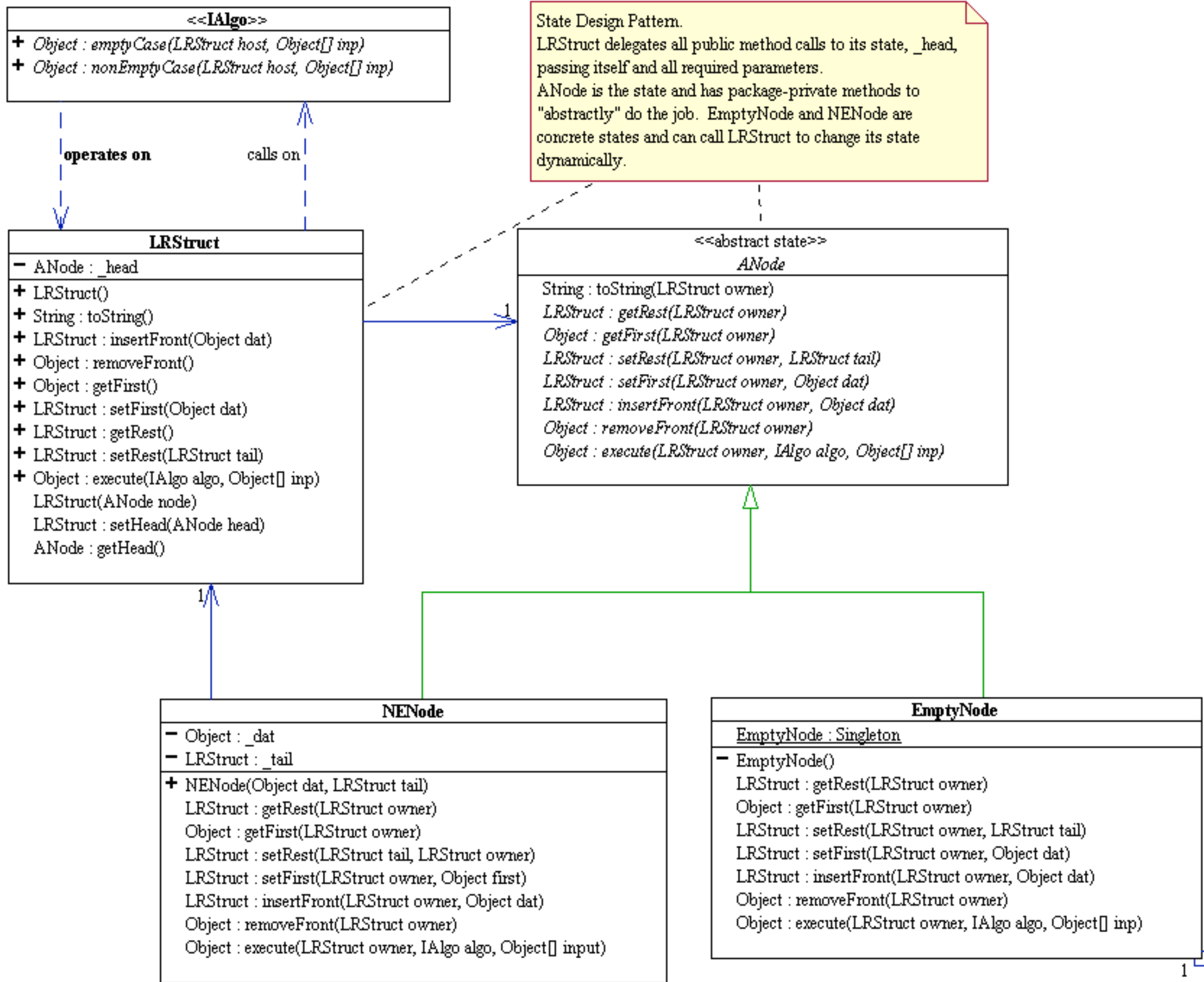
Please write and sign the Rice Honor Pledge to affirm that you had a valid reason why you were unable to take the scheduled exam: "On my honor, I had a valid reason for not taking the scheduled exam."

1a	1b	1c	2	Total
/40 pts	/10 pts	/10 pts	/40 pts	/100 pts

Reference: Immutable List Framework (`IList`)

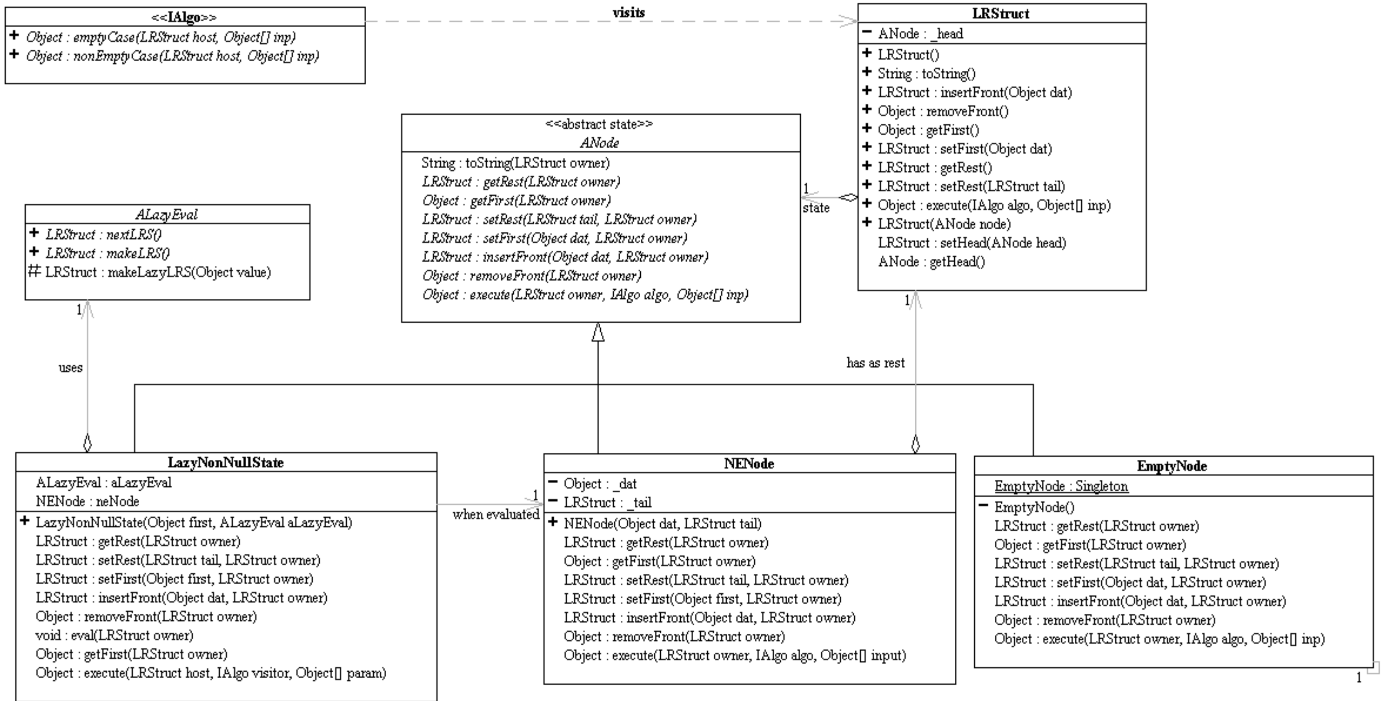


Reference: Mutable List Framework (`LRStruct`)

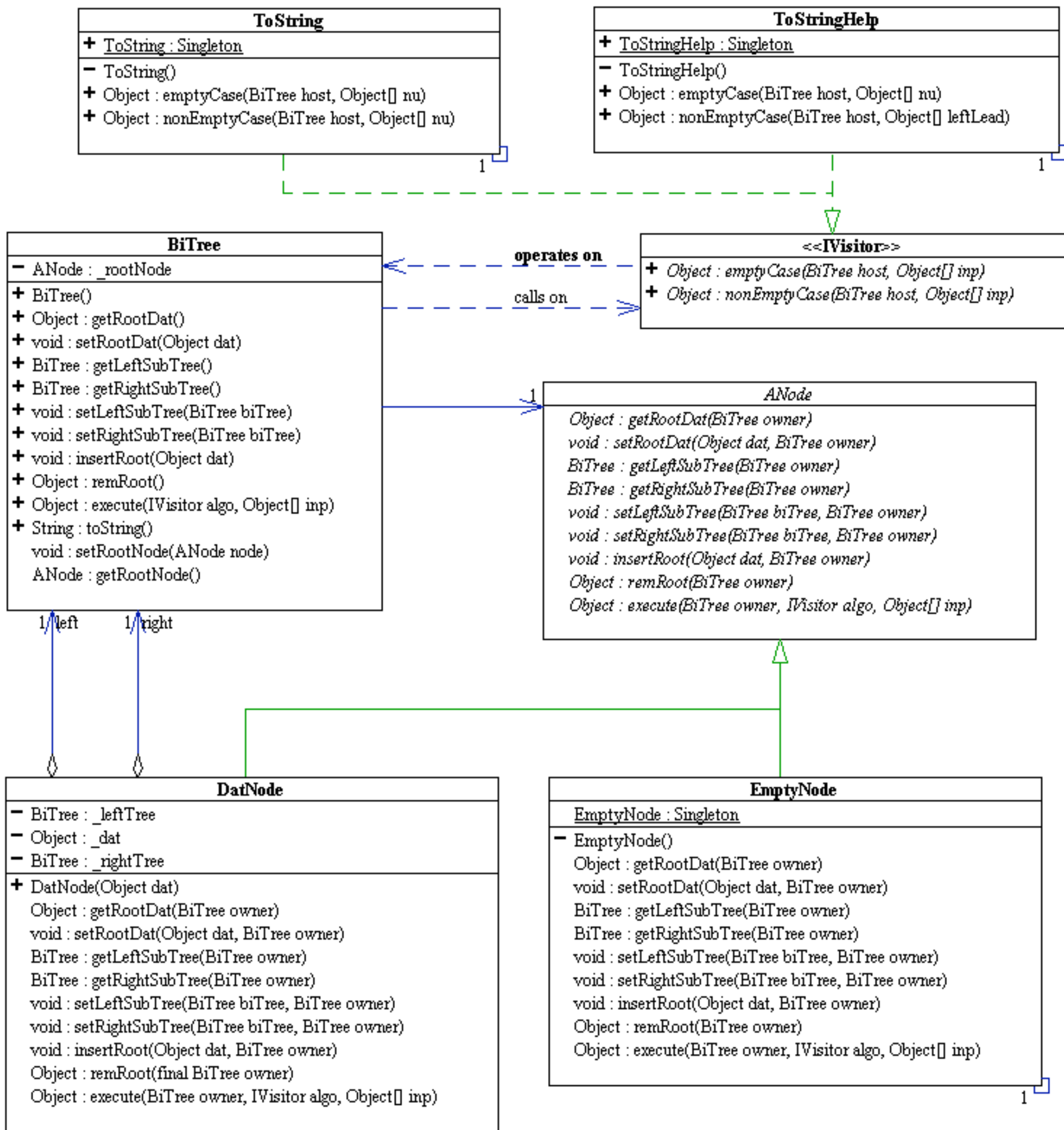


NAME & ID#: ______________________________

Reference: Lazy Mutable List Framework (`LRStruct` with `ALazyEval`)



Reference: Binary Tree Framework (`BiTree`)



Lazy Evaluation:

a) (40 pts) **You are to write an `ALazyEval` called `LazyLambdaApplyEval`, which takes in an `LRStruct` filled with `ILambdas` and an `LRStruct` with other values, that will create a possibly infinite list containing the result of applying the lambda in the front of the lambdas list to the values list.**

- If the values list is empty, the list created by `LazyLambdaApplyEval` is empty.
- If the lambdas list is empty, the list created contains the first element in the values list as first. The rest is created lazily using the rest of the non-empty list.
- If both input lists are non-empty, `LazyLambdaApplyEval` removes the lambda from the front of the lambdas list. It then calls the lambda with the values list as argument and returns the result of the lambda. The rest is created lazily from the rests of the two lists.
- Example:
  - lambda: "remove the first from the values lists and return the square"
  - lambdas = (), values = () → result = ()
  - lambdas = (lambda), values = () → result = ()
  - lambdas = (), values = (1) → result = (1)
  - lambdas = (lambda), values = (10) → result = (100)
  - lambdas = (lambda), values = (10, 11, 12) → result = (100, 11, 12)
  - lambdas = (lambda, lambda), values = (10, 11, 12, 13, 14) → result = (100, 121, 12, 13, 14)
  - lambdas = (lambda, lambda), values = (10) → result = (10)
  - lambdas = (lambda, lambda, …), values = (10, 10, 10, …) → result = (100, 100, 100, …)
- Example:
  - lambda(*n*): "remove *n* integers from the list and return the sum"
  - lambdas = (), values = () → result = ()
  - lambdas = (lambda(2)), values = () → result = ()
  - lambdas = (), values = (1) → result = (1)
  - lambdas = (lambda(2)), values = (2) → result = (2)
  - lambdas = (lambda(2)), values = (2, 3, 4) → result = (2+3, 4)
  - lambdas = (lambda(2)), values = (2, 3, 4, 5) → result = (2+3, 4, 5)
  - lambdas = (lambda(2), lambda(3), …), values = (2, 3, 4, 5, 6) → result = (2+3, 4+5+6)
  - lambdas = (lambda(2), lambda(3), …), values = (2) → result = (2)
  - lambdas = (lambda(2), lambda(3), …), values = (1, 1, 1, …) → result = (2, 3, 4, …)

Notes:

a. Think delegation always.
b. There are no `if` statements needed, unless you are working with primitive data (ints, doubles, etc.)!
c. Don't forget about the possibility that the source lists may be or become empty.

Please insert your code for `LazyLambdaApplyEval.java` below:

```
package lrs.lazyLRSEvaluators;

import lrs.*;
import fp.*;

/**
 * Lazily creates a possibly infinite list that contains the result of
 * applying a lambda from one input list to a value from a second input list.
 * If the list with the lambdas ends, the created list will contain just the
 * elements in the value list. If the value list ends, this list will end.
 */
```

```
public class LazyLambdaApplyEval extends ALazyEval {
    /** Constructs a LazyLambdaApplyEval, which creates a list from a list of
      * lambdas and a list of values. The lambda determines how the value list
      * is processed and which elements are "consumed", i.e. removed from the
      * front of the value list.
      * The lambda is always removed from the front of the lambda list.
      * @param src1 input list with lambdas
      * @param src2 input list with values
      */
    public LazyLambdaApplyEval(LRStruct lambdas, LRStruct values) {
        this.lambdas = lambdas;
        this.values = values;
    }

    // FOR STUDENT TO COMPLETE: add fields and methods here
    private LRStruct lambdas;
    private LRStruct values;

    public LRStruct nextLRS() {
        return makeLRS();
    }

    /**
     * If the lambda list is non-empty, remove the lambda a lambda and pass the value list
     * to it. Return the value of the lambda application. If the lambda list is empty,
     * clone the value list.
     * @return a list with the elements that come out of the lambdas.
     */
    public LRStruct makeLRS() {
        return (LRStruct)lambdas.execute(new IAlgo() {
            public Object emptyCase(LRStruct host, Object... inp) {
                return values.execute(new IAlgo() {
                    public Object emptyCase(LRStruct host, Object... inp) {
                        return new LRStruct(); // both empty
                    }
                    public Object nonEmptyCase(LRStruct host, Object... inp) {
                        return makeLazyLRS(values.removeFront());
                    }
                });
            }
            public Object nonEmptyCase(LRStruct host, Object... inp) {
                final ILambda l = (ILambda)lambdas.removeFront();
                return values.execute(new IAlgo() {
                    public Object emptyCase(LRStruct host, Object... inp) {
                        return new LRStruct(); // both empty
                    }
                    public Object nonEmptyCase(LRStruct host, Object... inp) {
                        return makeLazyLRS(l.apply(values));
                    }
                });
            }
        });
    }
}
```

1. **Lazy Evaluation**:

b) (10 pts) **Write a subclass of the `LazyLambdaApplyEval` above called `LazySquareEval` and provide an infinite list of an `ILambda` that squares its input. `LazySquareEval` therefore creates a potentially infinite list with the squares of the values in the input list.**

- Example:
  - lambda: "remove the first from the values lists and return the square"
  - lambdas = (lambda, lambda, …)
  - values = () → result = ()
  - values = (2) → result = (4)
  - values = (2, 3) → result = (4, 9)
  - values = (10, 10, 10, …) → result = (100, 100, 100, …)

Please insert your code for `LazySquareEval.java` below:

```
package lrs.lazyLRSEvaluators;

import fp.*;
import lrs.*;

/**
 * Lazily creates a possibly infinite list that squares the Integers in
 * an input list.
 */
public class LazySquareEval extends LazyLambdaApplyEval {
    // FOR STUDENT TO COMPLETE: add fields and methods here
    public LazySquareEval(LRStruct src) {
        super(new ALazyEval() {
            final ILambda squareLambda = new ILambda() {
                public Object apply(Object... p) {
                    LRStruct l = (LRStruct)p[0];
                    Integer i = (Integer)l.removeFront();
                    return i;
                }
            };
            public LRStruct makeLRS() {
                return makeLazyLRS(squareLambda);
            }
            public LRStruct nextLRS() {
                return makeLRS();
            }
        }.makeLRS(),src);
    }
}
```

1. **Lazy Evaluation**:

c) (10 pts) **Write a subclass of the `LazyLambdaApplyEval` above called `LazyIncreasingAdderEval` and provide an infinite list of `ILambda`, where the first ILambda removes two numbers from the values list and returns their sum, the second three numbers, and so on – the *n*th ILambda removes *n+1* numbers from the values list and returns their sum.**

- Example:
  - lambda(*n*): "remove *n* integers from the list and return the sum"
  - lambdas = (lambda(2), lambda(3), …)
  - values = () → result = ()
  - values = (1) → result = (1)
  - values = (1) → result = (1)
  - values = (1, 2) → result = (1+2)
  - values = (1, 2, 3) → result = (1+2, 3)
  - values = (1, 2, 3, 4) → result = (1+2, 3+4)
  - values = (1, 2, 3, 4, 5) → result = (1+2, 3+4+5)
  - values = (1, 2, 3, 4, 5, 6) → result = (1+2, 3+4+5, 6)
  - values = (1, 1, 1, …) → result = (2, 3, 4, …)

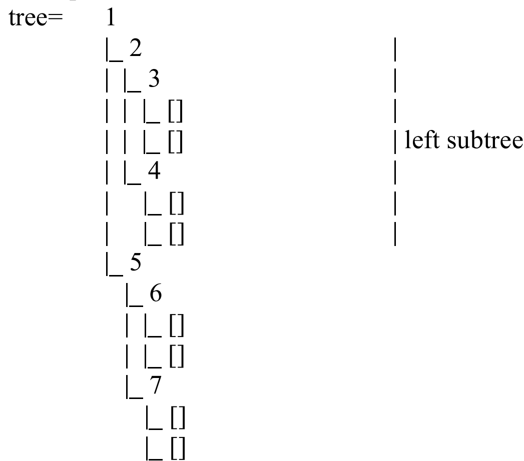
Please insert your code for `LazyIncreasingAdderEval.java` below:

```
p package lrs.lazyLRSEvaluators;

import fp.*;
import lrs.*;

/**
 * Lazily creates a possibly infinite list that sums groups of Integers
 * in the input list. Initially it adds two Integers, then three Integers,
 * then four, etc.
 * If the list becomes empty at any time, the generated list ends.
 */
public class LazyIncreasingAdderEval extends LazyLambdaApplyEval {
    // FOR STUDENT TO COMPLETE: add fields and methods here
    static class AdderLambda implements ILambda {
        int num;
        public AdderLambda(int n) { num = n; }
        public Object apply(Object... p) {
            final LRStruct l = (LRStruct)p[0];
            return l.execute(new IAlgo() {
                public Object emptyCase(LRStruct host, Object... inp) {
                    return 0;
                }
                public Object nonEmptyCase(LRStruct host, Object... inp) {
                    if (num--==0) { return 0; }
                    Integer i = (Integer)l.removeFront();
                    return i+(Integer)(l.execute(this));
                }
            });
        }
    }

    public LazyIncreasingAdderEval(LRStruct src) {
        super(new ALazyEval() {
            int num = 2;
            public LRStruct makeLRS() {
                return makeLazyLRS(new AdderLambda(num++));
            }
            public LRStruct nextLRS() {
                return makeLRS();
            }
        }.makeLRS(),src);
    }
}
```

2. **Binary Trees**:

(40 pts) **You are to write an `IVisitor` called `BRSClipRight` that will create an `LRStruct` containing the root at the beginning, and the nodes in the right subtree in an pre-order fashion. The tree then "becomes" the left subtree.**

- If the tree is empty, the list returned is empty, and the tree remains empty.
- If the tree is non-empty, the list returned contains the element in the root of the tree, and all elements in the nodes of the right subtree in a pre-order fashion. The tree then "becomes" the left subtree:
  - If the left subtree is empty, the tree becomes empty.
  - If the left subtree is non-empty, the tree's root, left and right subtree get set to the left subtree's root, left and right subtree.
- Example:



```
tree=   1
        |_ 2                |
        | |_ 3              |
        | | |_ []           |
        | | |_ []           | left subtree
        | |_ 4              |
        |   |_ []           |
        |   |_ []           |
        |_ 5
          |_ 6
          | |_ []
          | |_ []
          |_ 7
            |_ []
            |_ []
```

clipRight, list returned is (1 5 6 7)
and the tree is changed to

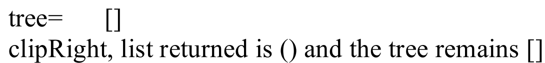
```
tree=   2
        |_ 3                |
        | |_ []             |
        | |_ []             | left subtree
        |_ 4                |
```

(This constitutes one application of the BRSClipRight visitor)

clipRight, list returned is (2 4)
and the tree is changed to

```
tree=   3
        |_ []               | left subtree
        |_ []
```

clipRight, list returned is (3)
and the tree is changed to



```
tree=   []
```

clipRight, list returned is () and the tree remains []

Notes:

a. Think delegation always.
b. There are no `if` statements needed!
c. Don't forget about the possibility that the tree may be or become empty.

Please insert your code for `BRSClipRight.java` below:

```
package brs.visitor;

import brs.*;
import lrs.*;

/**
 * This visitor returns an LRStruct that contains the root and all the nodes in
 * the right subtree in a pre-order fashion, and makes the left subtree this tree.
 **/
public class BRSClipRight implements IVisitor {
    public static final BRSClipRight Singleton = new BRSClipRight();
    private BRSClipRight() {}

    // STUDENT TO COMPLETE
    public Object emptyCase(BiTree host, Object... nu) {
        return new LRStruct();
    }

    public Object nonEmptyCase(final BiTree outerHost, Object... nu) {
        LRStruct lrs = (LRStruct)outerHost.getRightSubTree().execute(new IVisitor() {
            public Object emptyCase(BiTree host, Object... inp) {
                return inp[0];
            }
            public Object nonEmptyCase(BiTree host, final Object... inp) {
                LRStruct l = (LRStruct)host.getLeftSubTree().execute(this,
                                  host.getRightSubTree().execute(this, inp));
                l.insertFront(host.getRootDat());
                return l;
            }
        }, new LRStruct());
        lrs.insertFront(outerHost.getRootDat());
        outerHost.getLeftSubTree().execute(new IVisitor() {
            public Object emptyCase(BiTree host, Object... inp) {
                 // make this tree empty
                outerHost.setLeftSubTree(new BiTree());
                outerHost.setRightSubTree(new BiTree());
                outerHost.remRoot();
                return null;
            }
            public Object nonEmptyCase(BiTree host, final Object... inp) {
                outerHost.setRootDat(host.getRootDat());
                outerHost.setLeftSubTree(host.getLeftSubTree());
                outerHost.setRightSubTree(host.getRightSubTree());
                return null;
            }
        });
        return lrs;
    }
}
```