Comp 212 - Intermediate Programming EXAM #2 March 30, 2005
Rice University - Instructors: Cox & Nguyen NAME: _____

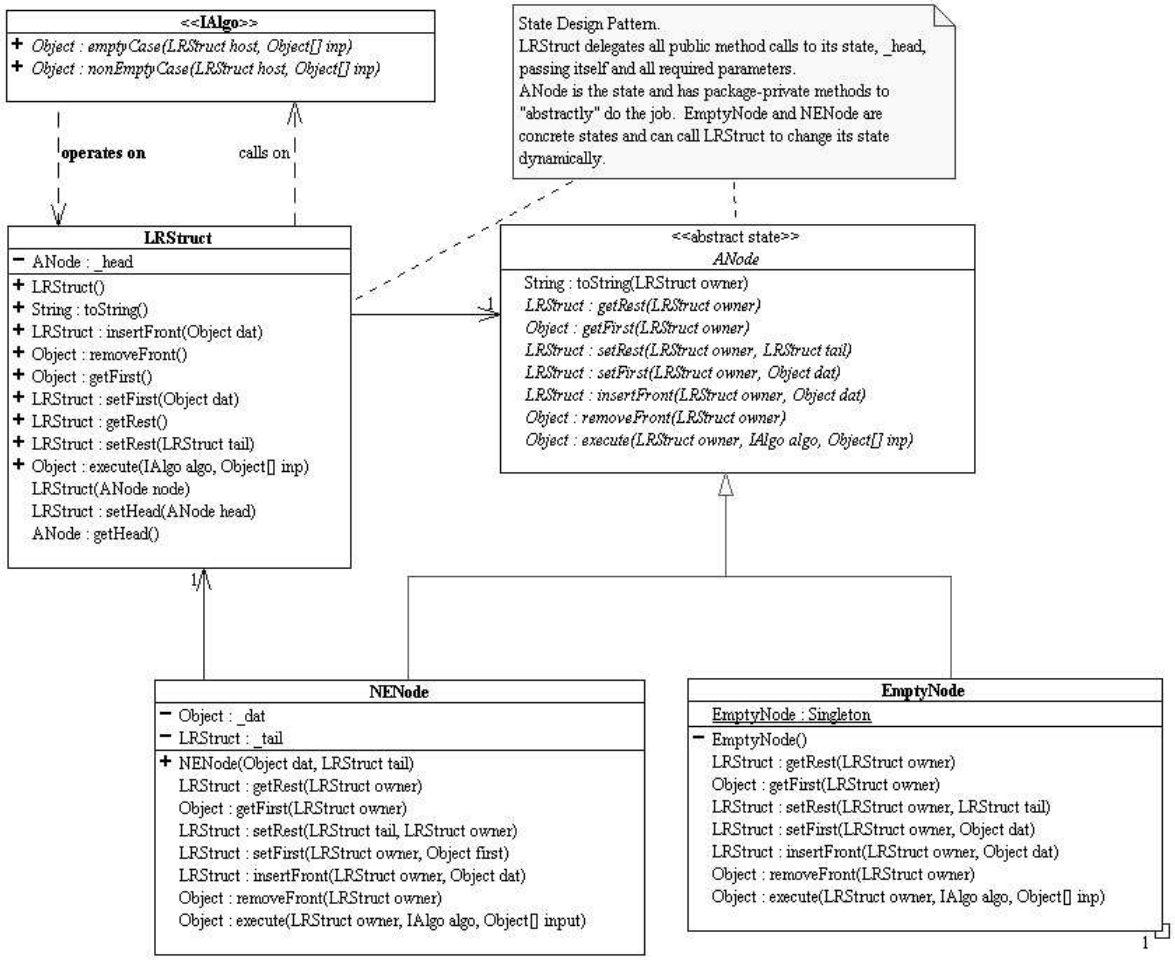
Instructions

1. This exam is conducted under the Rice Honor Code. It is a open-book 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. Make sure you use the Singleton pattern whenever appropriate. Unless specified otherwise, you do not need to write any code for it. Just write "singleton pattern" as a comment.
8. 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.
9. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1a) 5 pts	1b) 12 pts	1c) 8 pts	2a) 15 pts	2b) 10 pts	3)25 pts	4) 25 pts	Total 100 pts

For your convenience, below is the UML class diagram for the mutable list framework, LRStruct,studied in class. You are free to use this list framework without explanation/implementation.



1. Selection Sort Algorithm on LRStruct

a/ The following is an LRStruct visitor that removes the minimum element from an LRStruct containing Integer objects.

```
/**
 * Removes the minimum Integer element from an LRStruct containing Integers.
 */
public class RemMinLRS implements IAlgo {
    public static final RemMinLRS Singleton = new RemMinLRS();
    private RemMinLRS() {
    }
    /**
     * There is nothing to remove: throw an exception!
     */
    public Object emptyCase(LRStruct host, Object... nu) {
        throw new IllegalArgumentException("Empty host has no data.");
    }
    /**
     * Calculates the minimum and passes the list that contains the accumulated
     * minimum down to the rest for it to "help" find the minimum.
     * The input parameter for the anonymous helper is the list whose first is
     * the smallest element that precedes the host parameter.
     */
    public Object nonEmptyCase(LRStruct host, Object... nu) {
        return host.getRest().execute(new IAlgo() {
            /**
             * We are at the end of the list.  Since acc is the LRStruct whose
             * first is the smallest element that precedes the host h, just
             * tell acc to remove its first.
             * @param acc acc[0] an LRStruct whose first is the smallest element that
             * precedes the host h.
             */
            public Object emptyCase(LRStruct h, Object... acc) {
                return ((LRStruct)acc[0]).removeFront();
            }
            /**
             * We still need to traverse the list to find the minimum.
             * Update the list containing the minimum, pass it down to the rest
             * and recurse.
             * @param acc acc[0] is an LRStruct whose first is the smallest element that
             * precedes the host h.
             */
            public Object nonEmptyCase(LRStruct h, Object... acc) {
                int f = (Integer)h.getFirst();
                int m = (Integer)((LRStruct)acc[0]).getFirst();
                return f < m ?
                    h.getRest().execute(this, h):
                    h.getRest().execute(this, acc[0]);
            }
        }
        , host);
    }
}
```

Explain briefly how many comparisons between integers are needed to complete the task. Express the number of comparisons in terms of N, the number of elements in the host LRStruct.

b/ The following describes the selection sort algorithm on a list (LRStruct)

- When the list is empty, there is nothing to sort.
- When the list is non-empty:
  - Remove the minimum element of the list and insert it at the front of the list.
  - (Recursively) Sort the rest of the as-yet unsorted list.

Use RemMinLRS in 1a/ to write a visitor called SelectionSortLRS that sorts the host LRStruct according to the above selection sort algorithm.

```
// FILL IN YOUR CODE HERE
public class SelectionSortLRS implements IAlgo {
    public static final SelectionSortLRS Singleton = new SelectionSortLRS();
    private SelectionSortLRS() {
    }
    public Object emptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE



    }

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









    }
}
```

c/ Explain briefly how many comparisons between integers are needed to perform selection sort on a list. Express the number of comparisons in terms of N, the number of elements in the host LRStruct.

2. Merge Sort Algorithm on LRStruct.

a/ Write an LRStruct visitor called Unzip that removes the odd- indexed elements from the host LRStruct and returns these elements in another LRStruct. For example, suppose the host LRStruct is L = (a b c d e), then L.execute(Unzip.Singleton) returns the LRStruct (b d) and mutates the host list L into (a c e). (a).execute(Unzip.Singleton) returns an empty list and does not mutate the host list (a).

```
// FILL IN YOUR CODE HERE
public class Unzip implements IAlgo {
    public final static Unzip Singleton = new Unzip();
    private Unzip() {
    }
    /**
     * Do nothing.
     * @param nu not used
     * @return an empty LRStruct.
     */
    public Object emptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE


    }
    /**
     * @param nu not used
     * @return LRStruct containing the odd indexed element of the orignal host.
     */
    public Object nonEmptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE
        // HINT: If you recursively perform this algorithm on the rest of the host,
        // what do you get in return and what will happen to this rst?









    }
}
```

b/ The following describes the merge sort algorithm on a list.

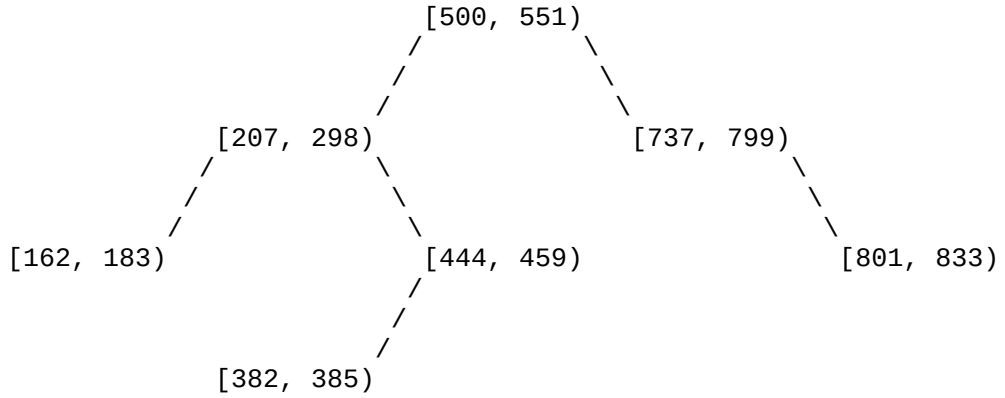
- Split the list into two sublists;
- (Recursively) Sort each of the sublists,
- Merge the sorted sublists into one list.

Assume you have a singleton LRStruct visitor called Merge that merges the host and input list into one list sorted in ascending order. Here the host and input list do not share any node and contain Integer objects sorted in ascending order. Also, they need not have the same length. In the end, both the host and the input list refer to the merged list. For example, (1 3 8 9).execute(Merge.Singleton, (-5 6 7)) will result in both the host and the input referencing (-5 1 3 6 7 8 9).

Use the results of 2a/ and the Merge visitor to write an LRStruct visitor to perform the above merge sort algorithm. Assume the host LRStruct contains Integers.

```
// FILL IN YOUR CODE HERE
public class MergeSort implements IAlgo {
    public final static MergeSort Singleton = new MergeSort();
    private MergeSort() {
    }

    /**
     * @param nu not used
     * @return null
     */
    public Object emptyCase(LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE


    }

    /**
     * @param nu not used
     * @return null
     */
    public Object nonEmptyCase(final LRStruct host, Object... nu) {
        // STUDENT TO COMPLETE
        // HINT: Use a helper.









    }
}
```

3. Consider a binary search tree (BST) in which the data are disjoint ranges, such as the following BST containing disjoint ranges of integers.



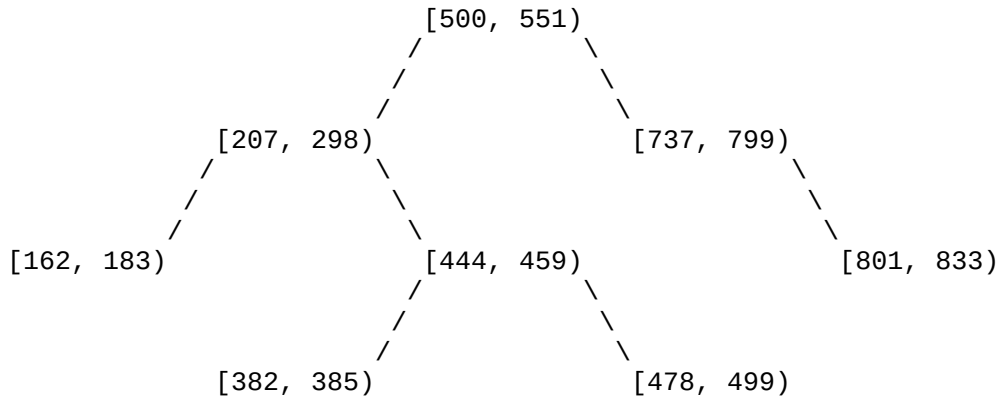
By disjoint, we mean that no two ranges overlap. Ranges are defined by a lower bound and an upper bound. The lower bound is included in the range. The upper bound is excluded from the range. In other words, the range [500, 551) includes the integers from 500 to 550; 551 is not included. Ranges of the form [x, x) are nonsensical and not allowed.

The data object representing a range implements the Comparable interface. The compareTo() method on the objects behaves as follows.
Consider [a, b).compareTo([c, d)). The compareTo() method returns the value of b - c.

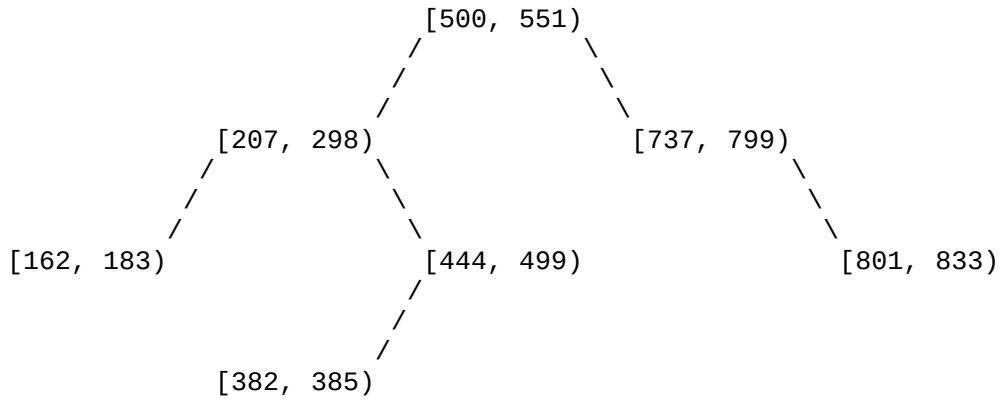
Write a visitor called BSTRangeInsert for inserting a new range into such a BST. This visitor must differ from the BSTInserter given in class in three ways:

- Obviously, your visitor must work with ranges; this requires changes to how data objects are compared.
- If a range is given to your visitor that overlaps with an existing range in the BST, you should do nothing and return null; otherwise, return the node that you insert or change, like the BSTInserter does. For example, inserting [451, 475) into the above tree would do nothing and return null because it overlaps with [444, 459).
- Your visitor must coalesce ranges whose upper and lower bounds are equal. In other words, the BST should never contain [a, b) and [b, c) as two distinct nodes. They should be coalesced as [a, c). See the hint below for an example of this situation.

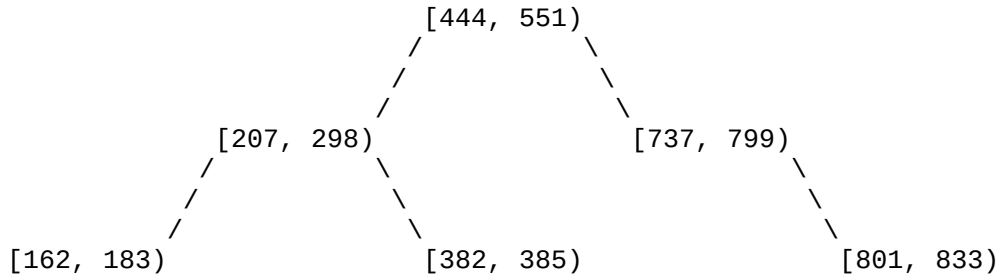
Hint: Your solution will have to handle three cases. First, consider the insertion of the range [478, 499) into the above tree. The resulting tree should be as follows.



Second, consider the insertion of the range [459, 499) into the tree.
The resulting tree should be as follows.



In other words, the ranges [444, 459) and [459, 499) are coalesced.
Third, consider the insertion of the range [459, 500) into the tree.
The resulting tree should be as follows.



In other words, the ranges [444, 459), [459, 500), and [500, 551) are coalesced.

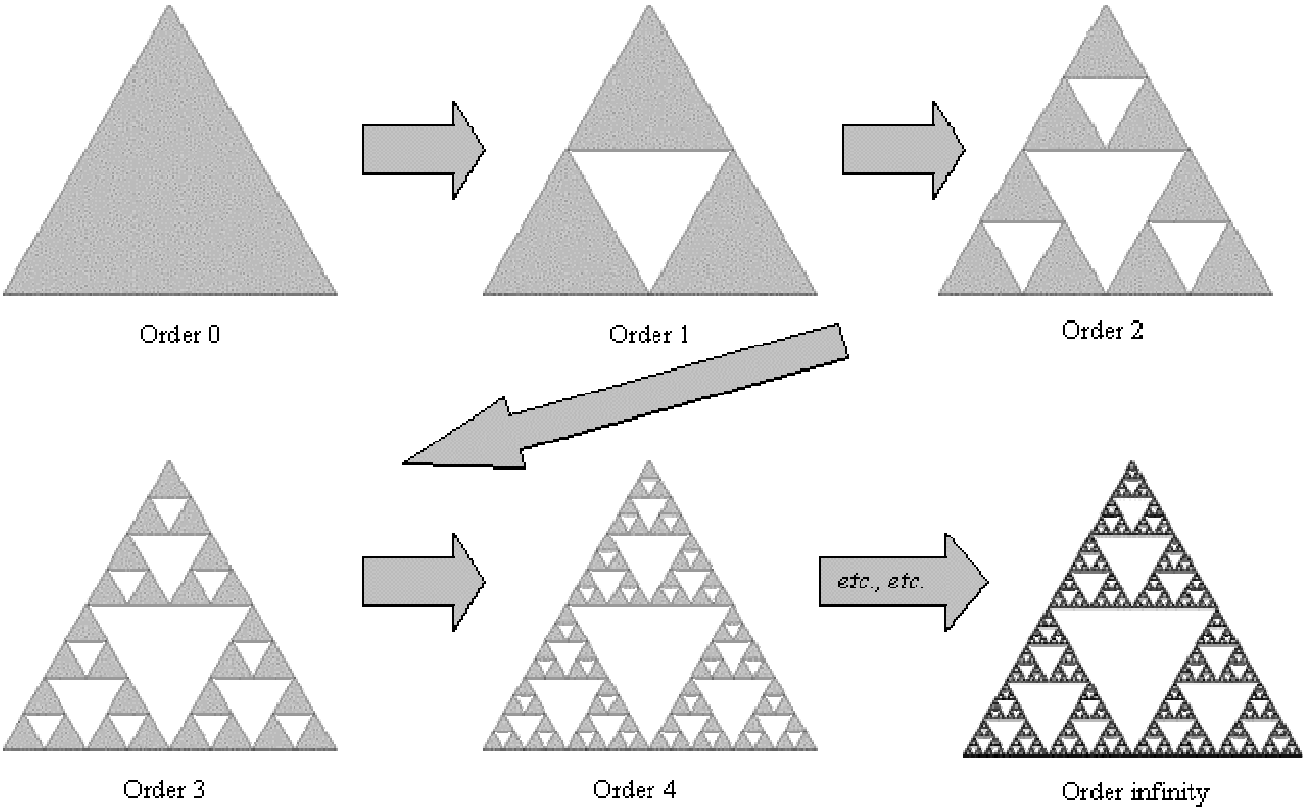
On the next page is the code for BST insertion algorithm that you may find helpful.

```
/**
* Inserts an Object into the host maintaining the host's binary search tree
* property via a given Comparator.
* Can also be used for Comparable objects.
* Duplication is not allowed: replaces old data object with the new one.
*/
public class BSTInserter implements IVisitor {
    private Comparator _order;
    /**
     * Used when the items in the tree are Comparable objects.
     */
    public BSTInserter() {
        _order = new Comparator() {
            public int compare(Object x, Object y) {
                return ((Comparable)x).compareTo(y);
            }
        };
    }
    /**
     * Used when we want to order the items according to a given Comparator.
     * @param order a total ordering on the items to be stored in the tree.
     */
    public  BSTInserter (Comparator order) {
        _order = order;
    }
    /**
    * Returns the host tree where the input is inserted as the root.
    * @param host an empty binary tree, which obviously satisfies BSTP.
    * @param input[0] new data to be inserted.
    * @return host (which is no longer empty).
    */
    public Object emptyCase(BiTree host, Object... input) {
        host.insertRoot (input[0]);
        return host;
    }
    /**
    * If the input is equal to host.getRootDat) then the old root data is
    * replaced by input.  Equality here is implicitly defined by the total
    * ordering represented by the Comparator _order.
    * @param host non-empty and satisfies BSTP.
    * @param input[0] new data to be inserted.
    * @return the tree where input[0] is inserted as the root.
    */
    public Object nonEmptyCase(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]);
        }
        host.setRootDat(input[0]);
        return host;
    }
}
```

Write the BSTRangeInsert visitor here.

4. Fractal Design

The following sequence of pictures depicts the growth at each step of special fractal called "Sierpinski Gasket."



Your task is to design an object model for the Sierpinski Gasket so that it can grow at each step as shown.

Express your design as Java classes/interfaces with appropriate data fields and methods. At the highest level your design should have a class called `Sierpinski` with at least a `grow` method that allows it to grow as shown. You need not have factories to grow since you are only growing in one specific way. You can directly manufacture a `Sierpinski` object by calling new on an appropriate constructor. Pseudo code in the form of comments is acceptable for the body of all methods.

Identify all design patterns used in your object model and explain the reasons why you are using them.

A blank page for the design