Comp 212 - Intermediate Programming EXAM #2 April 02, 2003
Rice University - Instructors: Cox & Nguyen NAME: _____

Instructions

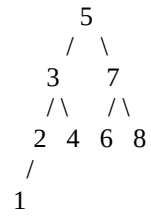
1. This exam is conducted under the Rice Honor Code. It is a closed-notes, closed-book exam.
2. Fill in your name on every page of the exam.
3. If you forget the name of a Java class or method, make up a name for it and write a *brief* explanation in the margin.
4. You are expected to know the syntax 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:

1) 25	2) 20	3) 25	4a) 15	4b) 15				TOTAL 100

1. Write a visitor for LRStruct called SplitHalf that breaks the LRStruct at the mid point, returning the LRStruct that begins at the mid point. E.g. when (1 2 3 4 5 6 7 8) executes this algorithm, it would become (1 2 3 4) and (5 6 7 8) is returned. If there are an odd number of elements, make the extra element a part of the rightmost list. Challenge (but not required): do it without computing the length. For example, the list (99) would become () and (99) will be returned.

2. Definition: An empty binary tree is *balanced*. A non-empty binary tree is said to be *balanced* if both of its subtrees are *balanced* and their height difference is at most one. Given an ordered LRStruct of Comparable objects, construct a balanced BST BiTree containing the same objects. Recursively apply the solution to question 1 to build the tree without comparing any objects. (Thus, you cannot use the BSTInserter visitor because it compares objects.) For example, the list (1 2 3 4 5 6 7 8) should produce:



3. Consider the following extension to the `IDictionary` interface (see attached).

```
public interface IRangeDictionary extends IDictionary {
    /**
     * Removes every DictionaryPair with a key greater than or equal
     * to keyLo and less than or equal to keyHi.  Returns an IList
     * of DictionaryPairs removed.  Throws an exception if keyLo is
     * greater than keyHi.
     */
    public IList removeRange(Comparable keyLo, Comparable keyHi);
}
```

Write the code for `removeRange` for the following binary search tree implementation of `IRangeDictionary`.

```
public class RangeDictBT implements IRangeDictionary {
    // A binary tree of DictionaryPairs ordered by key
    private BiTree _bt = new BiTree();

    // A trivial comparator for use by the binary search tree (BST) visitors
    private Comparator _comparator = new Comparator() {
          public int compare(Object x, Object y) {
              return ((Comparable)x).compareTo(y);
          }
    };

    // The visitors for basic binary search tree (BST) operations
    private IVisitor _finder = new BSTFinder(_comparator);
    private IVisitor _inserter = new BSTInserter(_comparator);
    private IVisitor _deleter = new BSTDeleter(_comparator);

    /**
     * code of methods for IDictionary elided…
     */

    public IList removeRange(Comparable keyLo, Comparable keyHi) {
          // for students to write.
    }
```

Note well: It may NOT be possible to efficiently enumerate every item between `keyLo` and `keyHi`. Therefore, your solution must not assume this. You are free to add any new private fields and methods.

4. java.util.Enumeration is an interface specified as follows.

```
public interface Enumeration {
    /**
    * Returns true if and only if this enumeration object contains at least one
    * more element to provide; false otherwise.
    */
    public boolean hasMoreElements();

    /**
     * Returns the next element of this enumeration if this enumeration object
     * has at least one more element to provide.
     * @exception NoSuchElementException - if no more elements exist.
     */
    public Object nextElement();
}
```

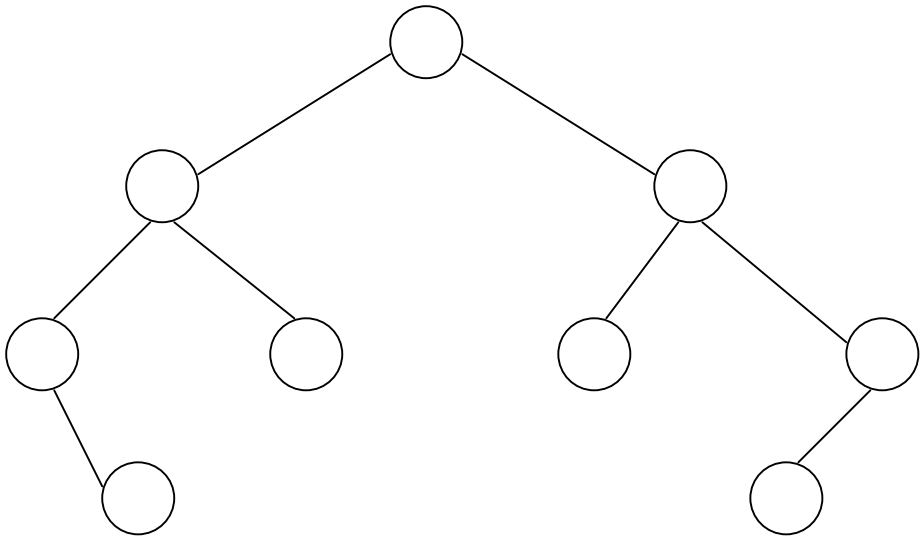
Consider the following traversal algorithm on a binary tree that also implements Enumeration. (In Java, a class may implement as many interfaces as desired; implementing an interface means providing concrete code for each of the interface's methods.)

```
import brs.*;  // binary tree structure
import rac.*;  // restricted access container
import java.util.Enumeration;

public class TreeTraversal implements IVisitor, Enumeration {
    private IRAContainer _rac;

    /**
     * @param tree != null.
     * @param rac != null.
     */
    public TreeTraversal(BiTree tree, IRAContainer rac) {
        _rac = rac;
        tree.execute(this, null);
    }

    public boolean hasMoreElements() {
        return !_rac.isEmpty();
    }

    public Object nextElement() {
        BiTree current = (BiTree) _rac.get();
        current.getRightSubTree().execute(this, null);
        return current.getRootDat();
    }

    public Object emptyCase(BiTree host, Object input) {
        return null;
    }

    public Object nonEmptyCase(BiTree host, Object input) {
        _rac.put(host);
        return host.getLeftSubTree().execute(this, input);
    }

}
```

Consider the following code fragment:

```
Enumeration e = new TreeTraversal(someTree, someRac);
while (e.hasMoreElements()) {
    System.out.println("" + e.nextElement());
}
```

a. Assume in the above code fragment that the `IRAContainer _someRac` is a stack; i.e. it is created by a call to `makeStack()` on some `IRACFactory` object. Label the nodes in the following tree diagram according to the order that the nodes are printed. In other words, the first node printed should be labeled 1, the second node printed should be labeled 2, etc.



b. Assume in the above code fragment that the `IRAContainer _someRac` is a queue; i.e. it is created by a call to `makeQueue()` on some `IRACFactory` object. Label the nodes in the following tree diagram according to the order that the nodes are printed. In other words, the first node printed should be labeled 1, the second node printed should be labeled 2, etc.