Comp 212 - Intermediate Programming EXAM #2 April 03, 2002
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. Use anonymous inner classes whenever appropriate.
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 may use all the visitors from the lectures and the homeworks without explanation/implementation.
10. You have two hours and a half to complete the exam.

Please State and Sign your Pledge:

1) 20	2) 20	3a) 10	3b) 15	4) 15	5) 20			TOTAL 100

1. (20 pts) Consider the following new container class, FixedSizeDict, implementing the *IDictionary* interface. FixedSizeDict differs from DictBST, DictLRS and DictArray discussed in the lectures in the following respects:

    a. It can hold at most a fixed number of key-value pairs (DictionaryPair). This number is passed to the constructor for FixedSizeDict.

    b. If the FixedSizeDict is full and the program attempts to insert a new key-value pair into the container, the *least recently accessed* key-value pair is dropped from the FixedSizeDict to make room. An ''access" to a key-value pair is either a lookup() for the key or the original insertion of the key-value pair into the FixedSizeDict.

    For example, suppose we create a FixedSizeDict of fixed size four, that is it can contain at most four key-value pairs. If we insert the key-value pairs (1,1), (2,2), (3,3), (4,4), and (5,5) in the given order, (1,1) will be dropped to make room for (5,5). If, however, we insert the key-value pairs (1,1), (2,2), (3,3), and (4,4), followed immediately by a lookup(1) and a lookup(2), then (3,3) will be dropped to make room for inserting (5,5).

    Specify FixedSizeDict 's data fields and write the code for the constructor, lookup() and insert() methods. You do not need to implement the other methods of the *IDictionary* interface. Your implementation must be based on LRStruct. You are free to use RemLast, the visitor that removes the last element of a host LRStruct discussed in class, without writing the code for it. Use anonymous inner classes whenever appropriate.

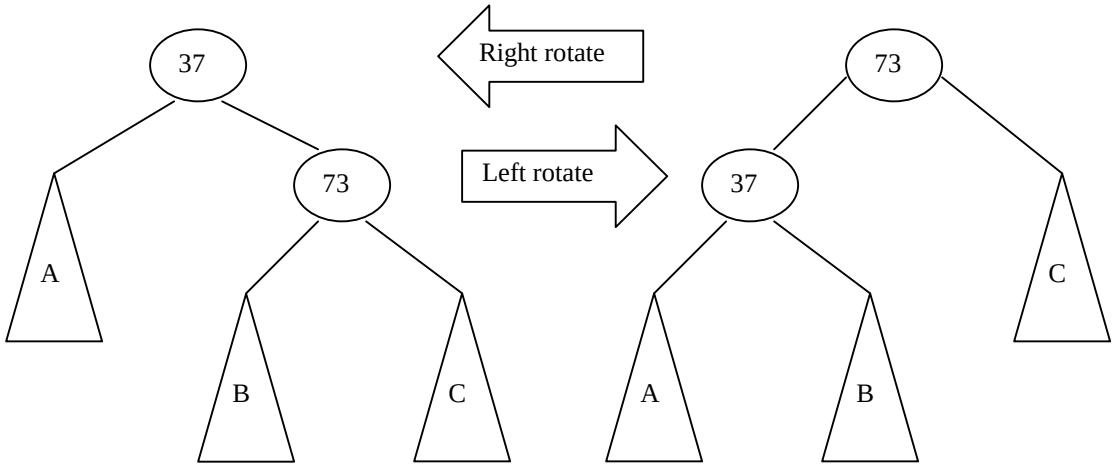
2. (20 pts) Given an LRStruct ("list") containing Integer objects, write a visitor called AddPairs to replace each pair of consecutive Integers in the host with their sum. For a host with an odd number of elements, the last element is left unchanged. For examples, (1 2 3 4) is transformed into (3 7), and (5 10 20 25 30 –30 17) is transformed into (15 45 0 17). Use anonymous inner classes whenever appropriate.

3. For the binary tree structure (BiTree) discussed in class, we allow the removal of the root only when both left and right subtrees are empty. In this problem, we want to relax this restriction and allow root removal when at least one of the subtrees is empty. Suppose that one of the subtrees is empty, then BiTree.remRoot() will change the state of the current BiTree to the state of the *other* subtree. For example, when the left subtree is empty, root removal of the parent tree will set the parent tree to its right subtree.

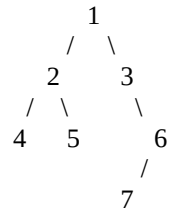
    a. (10 pts) Write the code for DatNode.remRoot (...).

b. (15 pts) Write a BiTree visitor, called BSTRemove, to delete a given *Comparable* input from the BiTree host with the binary search tree (BST) property. The algorithm must be written in a way that exploits the above specification of BiTree.remRoot(). Use anonymous classes whenever appropriate. You are free to use MaxTreeFinder and MinTreeFinder to obtain the subtrees containing the maximum and the minimum element, respectively.

4. (15 pts) The diagram below illustrates special operations on binary trees called *left rotation* and *right rotation*. These operations are inverses of each other. Write a visitor called LeftRotation to perform a left rotation on a BiTree host. Note that if the host tree is a BST then it is still a BST after a left (or right) rotation.



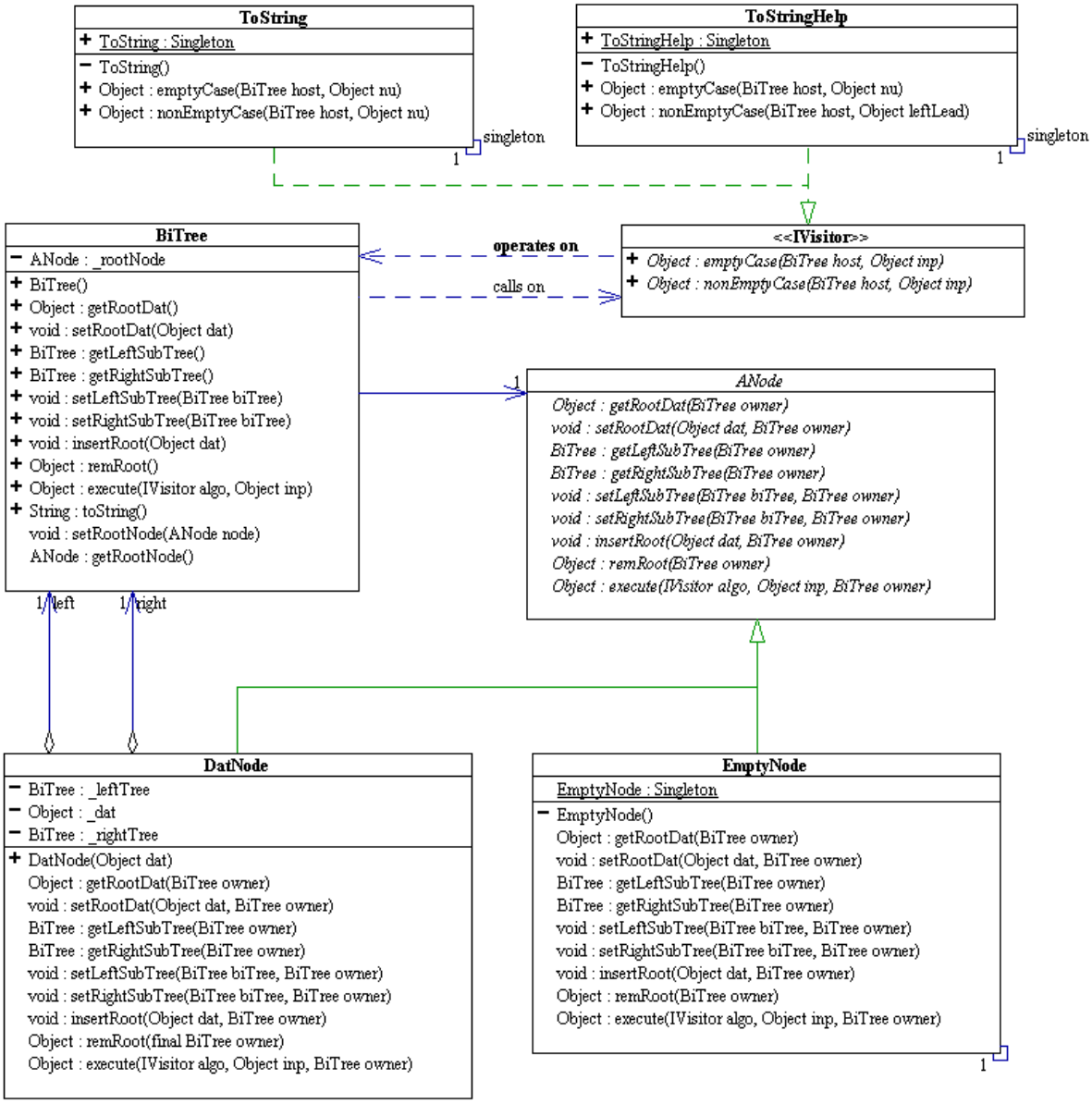
5. (20 pts) Define the *level* of a node in a tree as the number of branches (or edges) from the root node. For example, the level of the root node is 0. The level of the children trees of the root node is 1. Write a BiTree visitor, called BreadthTraversal, to print the tree nodes in breadth first traversal, that is the nodes at level n are printed from left to right before the nodes at level n+1 are printed. For example, the tree

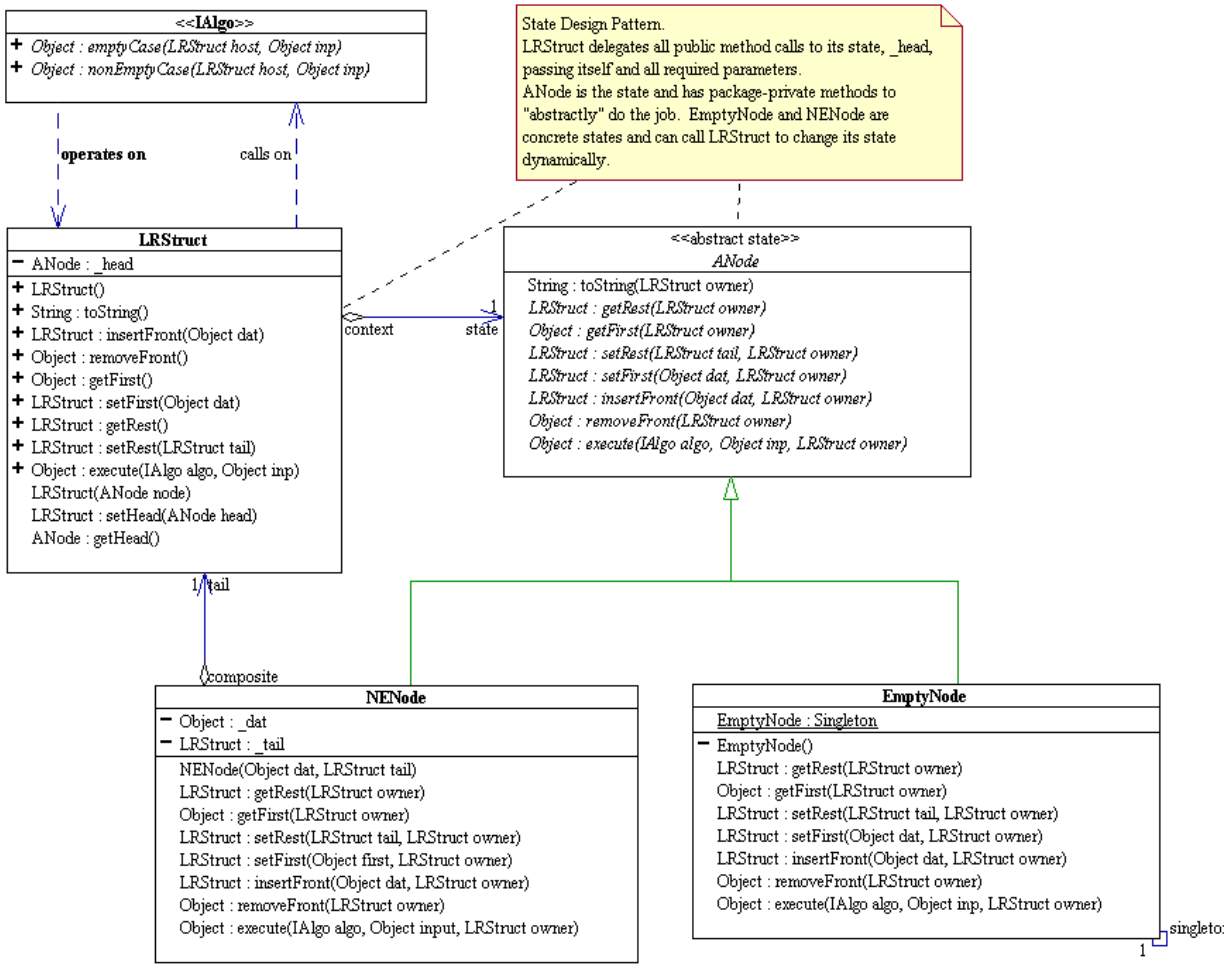


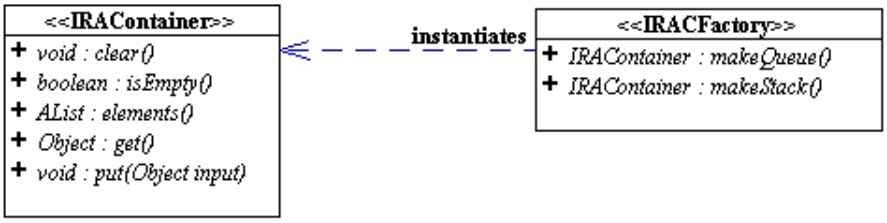
will print as 1 2 3 4 5 6 7.
The stub code below gives some hint as how to write the breadth first traversal algorithm.

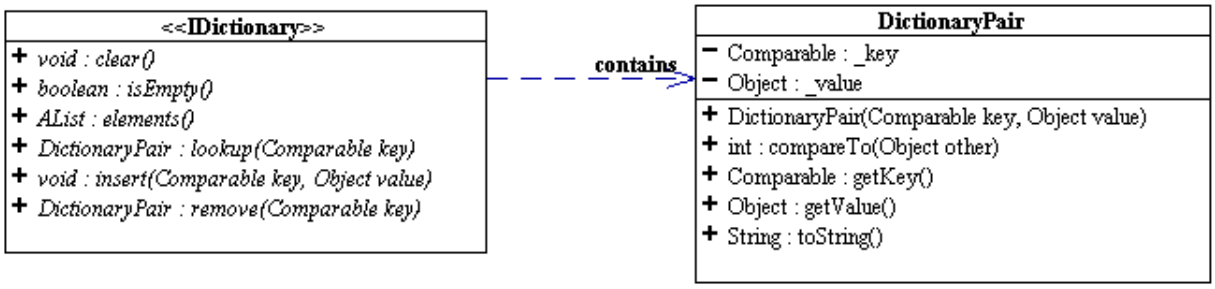
```
public class BreadthTraversal implements IVisitor {

    private IRAContainer _queue;  // first-in-first-out (FIFO) restricted access container
                                  // (called a queue) to save the subtrees that are yet to be
                                  // printed.
    public BreadthTraversal(IRACFactory qFac) {
        _queue = qFac.makeQueue();
    }

    /**
     * What to print here?
     */
    public Object emptyCase(BiTree host, Object nu) {
        // TO DO;




    }

    /**
     * Prints the root, saves the left and right subtrees in _queue.  Then process each element
     * in _queue as follows.
     * Get the front of the queue.  It's a tree!  There are two cases: empty or non-empty.
     * non-empty case: print the root, enter the left and right subtrees in the queue and recur.
     * empty case: what should we do here?  Think about it!
     */
    public Object nonEmptyCase(BiTree host, Object nu) {
        // TO DO;









    }
}
```

For your convenience, attached are the UML class diagrams for the binary tree, the mutable linear recursive structure, the dictionary, and the restricted access container studied in class. You are free to use their public interfaces without explanation/implementation.




<<IAlgo>>
+ Object : emptyCase(LRStruct host, Object inp)
+ Object : nonEmptyCase(LRStruct host, Object inp)
operates on
calls on
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.
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()
context
1
state
<<abstract state>>
ANode
String : toString(LRStruct owner)
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct tail, LRStruct owner)
LRStruct : setFirst(Object dat, LRStruct owner)
LRStruct : insertFront(Object dat, LRStruct owner)
Object : removeFront(LRStruct owner)
Object : execute(IAlgo algo, Object inp, LRStruct owner)
1
tail
composite
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(Object first, LRStruct owner)
LRStruct : insertFront(Object dat, LRStruct owner)
Object : removeFront(LRStruct owner)
Object : execute(IAlgo algo, Object input, LRStruct owner)
EmptyNode
EmptyNode : Singleton
- EmptyNode()
LRStruct : getRest(LRStruct owner)
Object : getFirst(LRStruct owner)
LRStruct : setRest(LRStruct tail, LRStruct owner)
LRStruct : setFirst(Object dat, LRStruct owner)
LRStruct : insertFront(Object dat, LRStruct owner)
Object : removeFront(LRStruct owner)
Object : execute(IAlgo algo, Object inp, LRStruct owner)
singleto
1


<<IRAContainer>>
+ void : clear()
+ boolean : isEmpty()
+ AList : elements()
+ Object : get()
+ void : put(Object input)
instantiates
<<IRACFactory>>
+ IRAContainer : makeQueue()
+ IRAContainer : makeStack()


<<IDictionary>>
+ void : clear()
+ boolean : isEmpty()
+ AList : elements()
+ DictionaryPair : lookup(Comparable key)
+ void : insert(Comparable key, Object value)
+ DictionaryPair : remove(Comparable key)
contains
DictionaryPair
- Comparable : _key
- Object : _value
+ DictionaryPair(Comparable key, Object value)
+ int : compareTo(Object other)
+ Comparable : getKey()
+ Object : getValue()
+ String : toString()