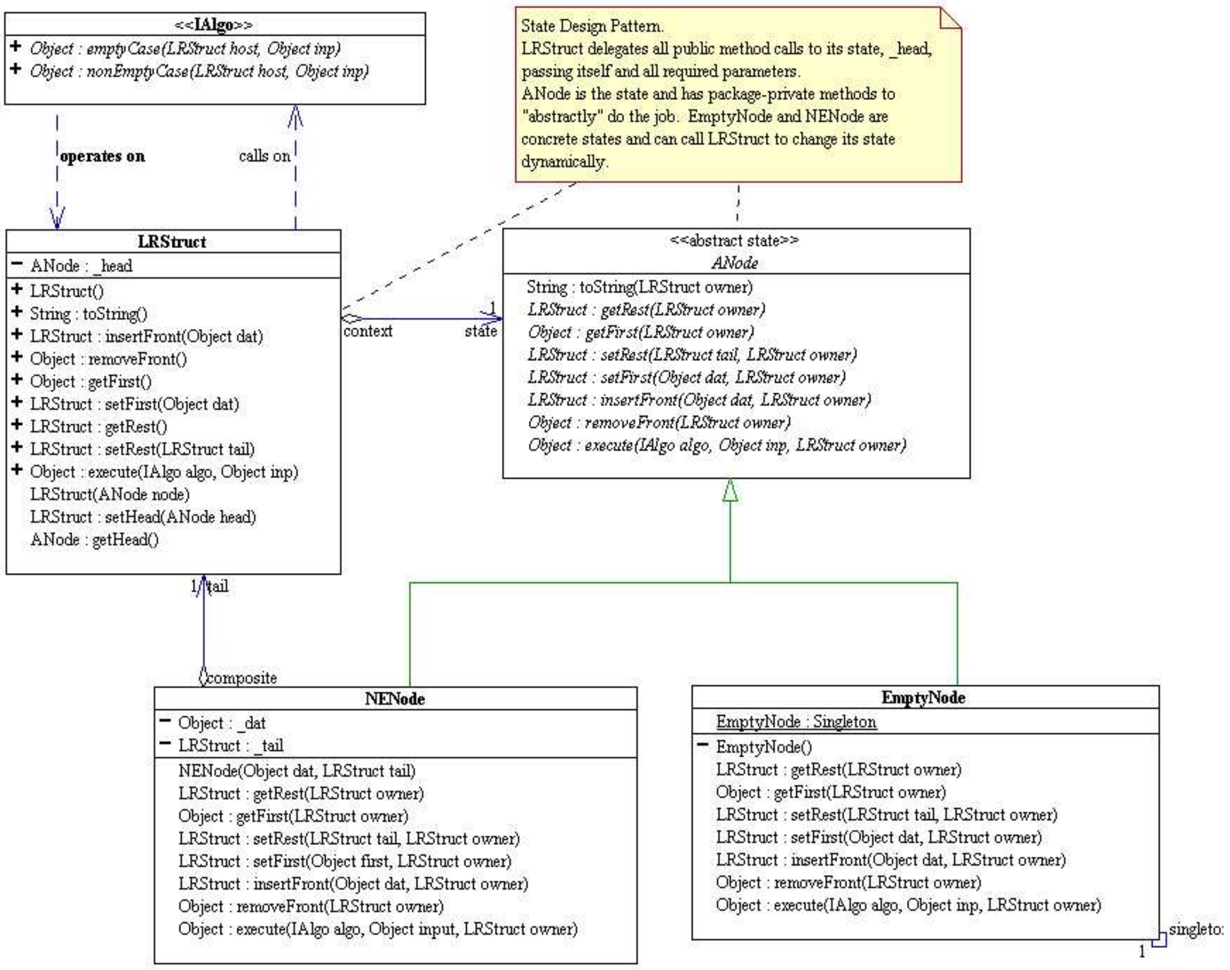
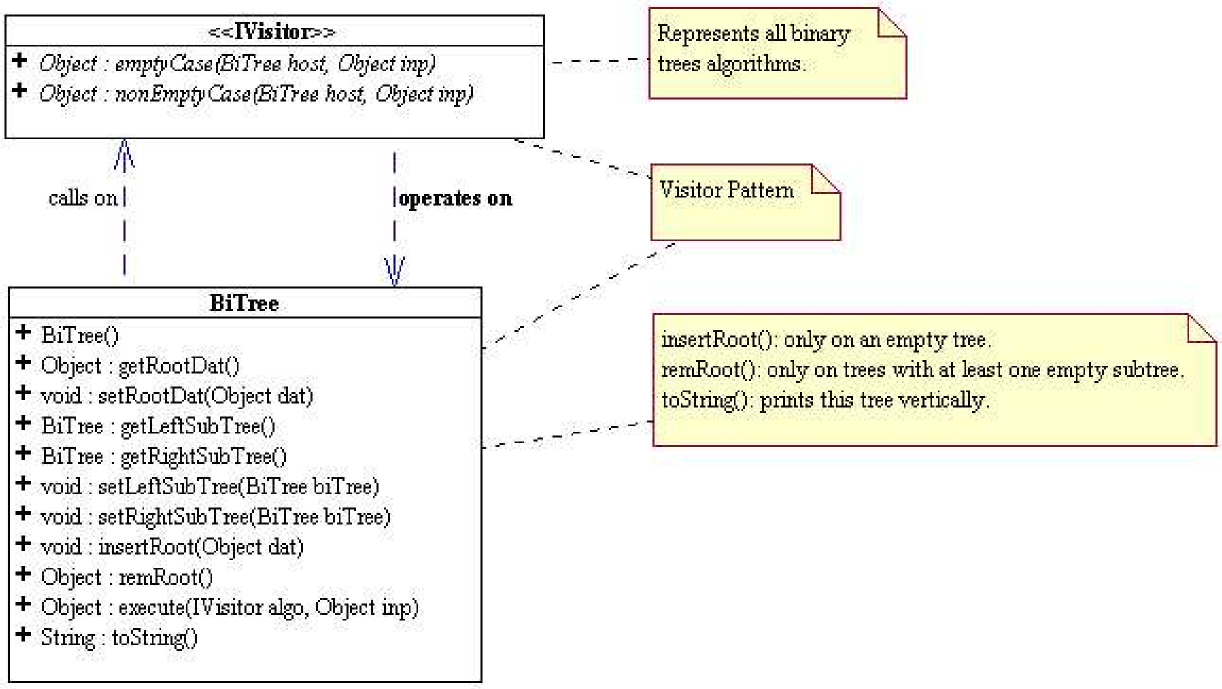
Comp 212 - Intermediate Programming   EXAM #2 Supplement   April 02, 2003
Rice University - Instructors: Cox & Nguyen

For your convenience, below is the UML class diagram for the mutable list framework LRStruct studied in class. You are free to use the public methods of IAlgo and LRStruct 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
state
1
<<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

Feel free to use the public methods of the BiTree and IVisitor


<<IVisitor>>
+ Object : emptyCase(BiTree host, Object inp)
+ Object : nonEmptyCase(BiTree host, Object inp)
Represents all binary trees algorithms.
Visitor Pattern
calls on
operates on
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.

Here are the IDictionary interface and the DictionaryPair class that you can freely use.

```
public interface IDictionary {
    /**
     * Clears the contents of the dictionary leaving it empty.
     */
    public void clear();

    /**
     * Returns true if the dictionary is empty and false otherwise.
     */
    public boolean isEmpty();

    /**
     * Returns an IList of DictionaryPairs corresponding to the entire
     * contents of the dictionary.
     */
    public IList elements();

    /**
     * Returns the DictionaryPair with the given key.  If there is not
     * a DictionaryPair with the given key, returns null.
     * Returns a DictionaryPair rather than the value alone so that
     * the user can distinguish between not finding the key and
     * finding the pair (key, null).
     */
    public DictionaryPair lookup(Comparable key);

    /**
     * Inserts the given key and value.  If the given key is already
     * in the dictionary, the given value replaces the key's old
     * value.
     */
    public void insert(Comparable key, Object value);

    /**
     * Removes the DictionaryPair with the given key and returns it.
     * If there is not a DictionaryPair with the given key, returns
     * null.
     */
    public DictionaryPair remove(Comparable key);
}


public class DictionaryPair implements Comparable {
    private Comparable _key;
    private Object     _value;

    public DictionaryPair(Comparable key, Object value) {
         _key   = key;
         _value = value;
    }

    public int compareTo(Object other) {
         return _key.compareTo(((DictionaryPair)other)._key);
    }

    public Comparable getKey() {
         return _key;
    }

    public Object getValue() {
         return _value;
    }

    public String toString() {
         return "(" + _key + "," + _value + ")";
    }
}
```

Here are the interfaces for restricted access containers that you can freely use.

```
package rac;

import listFW.*;

public interface IRAContainer {
    // Empty the container.
    public void clear();

    // Return TRUE if the container is empty; otherwise, return FALSE.
    public boolean isEmpty();

    // Return TRUE if the container is full; otherwise, return FALSE.
    public boolean isFull();

    // Return an immutable list of all elements in the container.
    public IList elements();

    // Remove the next item from the container and return it.
    public Object get();

    // Add an item to the container.
    public void put(Object input);
}
```

```
package rac;

public interface IRACFactory {
    // Create a ‘‘first-in, first-out’’ (FIFO) container.
    public IRAContainer makeQueue();

    // Create a ‘‘last-in, first-out’’ (LIFO) container.
    public IRAContainer makeStack();
}
```

Feel free to use the Becomes visitor given below.

Given two distinct LRStruct x and y, we can "assign" y to x by performing the following sequence of operations:

```
x.insertFront (null);
x.setRest (y);
x.removeFront();
```

x's head is now pointing to y's head.

We can codify the above as the following visitor.

```
/**
* Assigns the input list to the host.  host thus "becomes" the input list.
* In the end, both host and the input list share the same head node.
*/
public class Becomes implements IAlgo {
    public final static Becomes Singleton = new Becomes ();
    private Becomes() {
    }

    /**
     * @param input a LRStruct;
     * @return null
     */
    public Object emptyCase(LRStruct host, Object input) {
        return assign(host, (LRStruct)input);
    }

    /**
     * @param input a LRStruct;
     * @return null
     */
    public Object nonEmptyCase(LRStruct host, Object input) {
        return assign(host, (LRStruct)input);
    }

    private Object assign (LRStruct lhs, LRStruct rhs) {
        lhs.insertFront (null);
        lhs.setRest (rhs);
        lhs.removeFront();
        return null;
    }
}
```