`LRStruct`: An Enhanced `AList`

- `LRStruct` supports additional operations on the list structure:
  - `void setFirst(Object dat)`
  - `void setRest(LRStruct tail)`
  - `void insertFront(Object dat)`
  - `Object removeFront()`

LRStruct: The Implementation

```
public class LRStruct
{
  private ANode _head;
  ...
  public final void setFirst(Object data)
  {
    _head.setFirst(data, this);
  }
  ...
  public final void setRest(LRStruct tail)
  {
    _head.setRest(tail, this);
  }
  ...
```

- The State Pattern: *Delegate all requests made to the system ("*`LRStruct`*") to the current state instance.*

LRStruct: The Implementation (cont.)

```
...
public final void insertFront(Object data)
{
  _head.insertFront(data, this);
}
...
public final Object removeFront()
{
  return _head.removeFront(this);
}
```

- Note: We pass the LRStruct object ("this") to the state instance ("_head") so that it can change the LRStruct object's state using setHead().

EmptyNode: The Implementation

```
class EmptyNode extends ANode
{
  ...
  void setFirst(Object data, LRStruct owner)
  {
    throw new java.util.NoSuchElementException("Empty list has no first.");
  }
  ...
  void insertFront(Object data, LRStruct owner)
  {
    owner.setHead(new NonEmptyNode(data));
  }
  ...
```

- Note: `EmptyNode` (and `NonEmptyNode`) are package private.

NonEmptyNode: The Implementation

```
class NonEmptyNode extends ANode
{
  private Object _data;
  private LRStruct _tail;
  ...
  void setFirst(Object first, LRStruct owner)
  {
    _data = first;
  }
  ...
  void setRest(LRStruct tail, LRStruct owner)
  {
    _tail = tail;
  }
  ...
```

NonEmptyNode: The Implementation (cont.)

```
...
void insertFront(Object data, LRStruct owner)
{
  owner.setHead(new NonEmptyNode(data, new LRStruct(this)));
  /* Details:
     LRStruct coOwner = new LRStruct(this);
     NonNullNode newNode = new NonNullNode(dat, coOwner);
     owner.setHead(newNode);
     "old" style: owner._head = newNode - cannot be done.
   */
}
...
```

`NonEmptyNode`: The Implementation (cont.)

```
...
Object removeFront(LRStruct owner)
{
  owner.setHead(_tail.getHead()); // owner._head = _tail._head
  return _data;
}
...
```

`LRStruct`: Final Observations

- Note the implementation of multiple constructors in `LRStruct` and `NonEmptyNode`. They differ in two ways:
  1. The number and type of arguments.
  2. The visibility. In particular, note that `LRStruct` has both `public` and package private constructors.

- `LRStruct`'s methods are declared final, preventing a derived class from overriding them.

A General-Purpose Container

```
public interface IContainer {
  Object find(Object key);
   // If there is an object associated with key
   // then this object is returned else null is returned.
  Object remove(Object key);
   // Afterwards, find(key) returns null, and if there is
   // an object associated with key then this object is
   // returned else null is returned.
  void insert(Object key, Object value);
   // (key, value) is stored in this container with no
   // duplication and such that find(key) returns value.
}
```

A Proposed Implementation

```
public class LRStructContainer extends LRStruct
                               implements IContainer {
  ...
  Object find(Object key)
  {
    ...
```

- What are the pros and cons of such an approach?