Mutation and Bi-Directional Linked Lists

Corky Cartwright

Stephen Wong

Department of Computer Science

Rice University

Mutation: Succumbing to the Dark Side?

Four common problems in computing:

- Assume that we are repeatedly evaluating a method/function **m** often evaluating **m** on the same list of arguments. How can we avoid performing the same computation more than once?
- Assume we want to compute the number of a nodes in a tree data structure where nodes can be shared (the standard situation in OO programming with immutable data). How can we efficiently perform this computation?
- Perhaps the simplest data structure from the perspective of machine implementation is the array: a fixed-size list of elements that is allocated in contiguous machine memory where each element **e** is represented by a fixed size chunk of memory. The array was the *only* data structure in the original Fortran language. How can we create such structures using simple machine operations? How can we efficiently compute new ones?
- How can I represent cyclic linked structures (general graphs rather trees)?

The best solutions to these four problems all rely on data *mutation*.

Mutation: Definition

Mutation is rebinding a variable to a new value. What is a variable? A cell in computer memory containing a value such as an **int** or an (address of an) **Object**. Rebinding that variable destroys the former binding, replacing the contents of the memory cell (for the variable) with a new value.

Mutation is nearly non-existent in mathematics. We don't change numbers or functions; we simply construct new ones. Why? From the perspective of human thought, creating new values is much simpler. We don't have to remember what changes have been made to existing values and there is no extra cost incurred in creating new mathematical objects as opposed to changing existing ones.

In computation, the trade-offs are different. Mutation may have a large conceptual overhead--we have remember exactly what has changed at any point in a computation--but it also has huge efficiency and modeling advantages. The efficiency advantage is that the cost of creating a new data structure (assuming we can dispense with an existing one) is simply the cost of the changes (differences) between the new structure and the old one.

Modeling Involves Mutation

In many computational models, objects in the model evolve over time. Examples:

- Bank accounts
- Stock prices
- Enrollment (and roster) of a college class
- Temperature in your dorm room

Physical systems change over time, but the identities of the objects in the system change much less often that the properties of those objects. Example: humanity. Every few seconds, significant properties of almost every human being change (location, heart rate, posture, etc.) but new human beings are born infrequently (relative to changes in the status of the existing population).

Mutation Manifesto

Execution recapitulates system evolution

Given a physical system, it evolves in time. In most computations, the natural way to model this evolution is to simply update a data structure representing the state of the system.

What is the functional (immutable alternative)?

Modeling physical systems as functions mapping time to states. But this is expensive (and in many cases conceptually exhausting) because all history is explicitly retained in the computational model.

Side Effects: A Double-edged Sword

```
public class Box<T> {
  private T data;
  public Box(T data) { this.setData(data);}
  public T getData() {return data;}
  public void setData(T data) {this.data = data;}
}

Box<Integer> b1 = new Box<Integer>(42);  // create an instance of a Box.
Box<Integer> b2 = b1;  // b1 and b2 refer to the same Box containing "42"

b1.setData(123);   // mutate the Box's contents.

b2.getData() // now returns "123" even though b2 was never touched.
```

Very useful, but also very dangerous!

- *Useful because we can model something changing due to outside influences.*
- *Dangerous because one can never tell if or when something will change.*

OOP Mutation Guidelines

OO style dictates the disciplined use of mutation

- ***Never modify fields directly*** – always grab control of the mutating process.
- ***Support high level mutation via mutating methods*** – don’t allow low-level manipulations of the data, rather represent mutations as high-level operations inherent in the larger model of your system.
- ***Limit the scope of mutations*** – Limit the extent that side-effects can be seen by using various encapsulation techniques (scoping, visibility, etc.).

Example 1: Memo Functions

Consider a naive program to compute the Fibonacci function. How can we speed it up without any mathematical reworking of the problem?

Brute force speed-up:

```
class MyMath {
  static long fib(int n) {
    if (n <= 1) return 1;
    else return fib(n-1) + fib(n-2);
  }
}
```

Java syntax note: `static` = belongs to the class, not the individual object instances and thus is accessible to all instances of that class and to outsiders (if visible) directly from the class, e.g. `MyMath.fib(42)`

Memo Functions cont.

We can avoid re-computing **fib(n)** for a given value of **n** by maintaining a table recording all previously computed values. We will use a **HashMap** (a dictionary) for this purpose although we could easily use an expandable array to represent the table with less (a constant factor) execution overhead but more programming effort

```
import java.util.HashMap;
class BetterMath {
  static HashMap<Integer, Long> Fib = new HashMap<Integer, Long>();
  static long fib(int n) {
    if (n <= 1) return 1;
    else {
      Long cachedAnswer = Fib.get(n);
      if (cachedAnswer != null) return cachedAnswer;
      else {
        long newAnswer = fib(n-1) + fib(n-2);
        Fib.put(n, newAnswer);
        return newAnswer;
      }
    }
  }
}
```

Example 2: Counting Tree Nodes

Idea: we want to avoid counting a node more than once. How can we do this? When we start to visit a node, abort the visitation if node has "already been visited". How do we determine if a node has "already been visited"?

- Add a boolean "flag" field to our node representation initialized to false and mutate it to true when a node is visited.
  - Requires changing the node representation.
  - Boolean flags be cleared (requiring a tree traversal) before reuse.
- Add a **static HashSet<Node>** field to the **Node** class (or other convenient class) that holds the set of nodes that have been visited.
  - Less intrusive; node representation is unchanged.
  - Slightly more overhead. How is **HashSet** implemented?

Example 3: Initializing and Manipulating Arrays

Can arrays be incorporated in a functional language? Yes but they can only be used to hold immutable tables mapping 0 < **i** < **n** to some type **T**.

How can we create them? We need a primitive array construction operation that takes two arguments **n** and a function **f** mapping **int** to **T** that specifies the value **f(i)** of the **i**th array element.

How can we initialize arrays without using functions as data (alternatively, *only* using simple machine operations)? By allocating a block of memory (of proper size) and mutating the elements in that block. Use a loop (a special form of tail recursion) like the following:

```
    for (int i = 0; i < n; i++) a[i] = <some expression in i>;
```

Recall that **for** expands into a **while**.

Example 4: Cyclic Linked Structures

Josephus Problem: naive simulation of cannibals killing captives arranged in a circle.

"Ring Buffers" – used in digital delay and capture systems.

Regular infinite trees or lists (analogous to repeating decimals) are easily represented by lists extended to allow back pointers, that is, lists or trees that reference themselves. These are related to fractals, tilings (ala M.C. Escher), etc.

Background on Lists

Scheme lists and composite pattern lists in Java are internally represented using a linked list of **Cons** nodes. Each **Cons** node **N** is a chunk of memory containing a field **first** and a field **rest**. In each node **N**, these fields are the addresses of:

- the object **o** that is the first element in the list rooted at **N** and
- the **Cons** node **N'** representing the rest of the list rooted at **N'**.

In functional programming (Java programming with immutable objects), these fields are never modified after they are initialized. In imperative (mutable data) programming, they can be modified.

Mutation can be performed with discipline and taste. We will focus initially on the mutable generalization of composite lists.

Mutable Generalization of Functional Lists

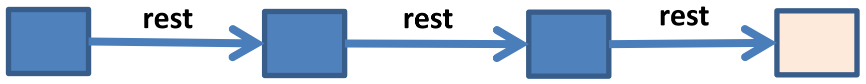
QuasiLists are singly-linked lists which make the **first** and **rest** fields of a **Cons** node mutable.   Nguyen and Wong's "Linear Recursive Structure" ("LRS") implementation also models the state transition that occurs when a list mutates between empty and non-empty.   See the class notes on OO design.

- The behavior of QuasiLists is essentially the same as immutable singly-linked lists, though the mutation can make certain operations faster by eliminating copying. Accessing any given data element is about the same speed however.
- QuasiLists do enable side-effects where decoupled parts of a system can share the effects of operations on a list of data.

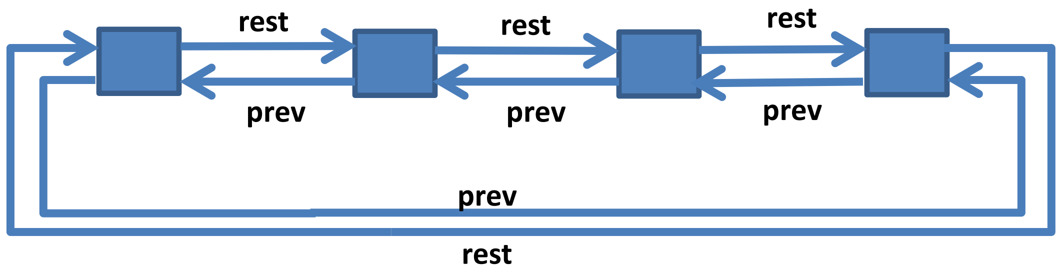
Doubly-linked Lists: BiLists

A “doubly-linked” list has two references to another list, one to the list *after* the current node (same as singly-linked list) and one to *previous* node.

Singly-linked:

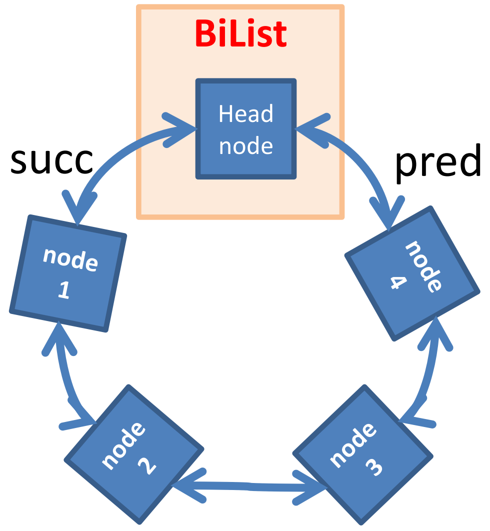


Doubly-linked & circular:

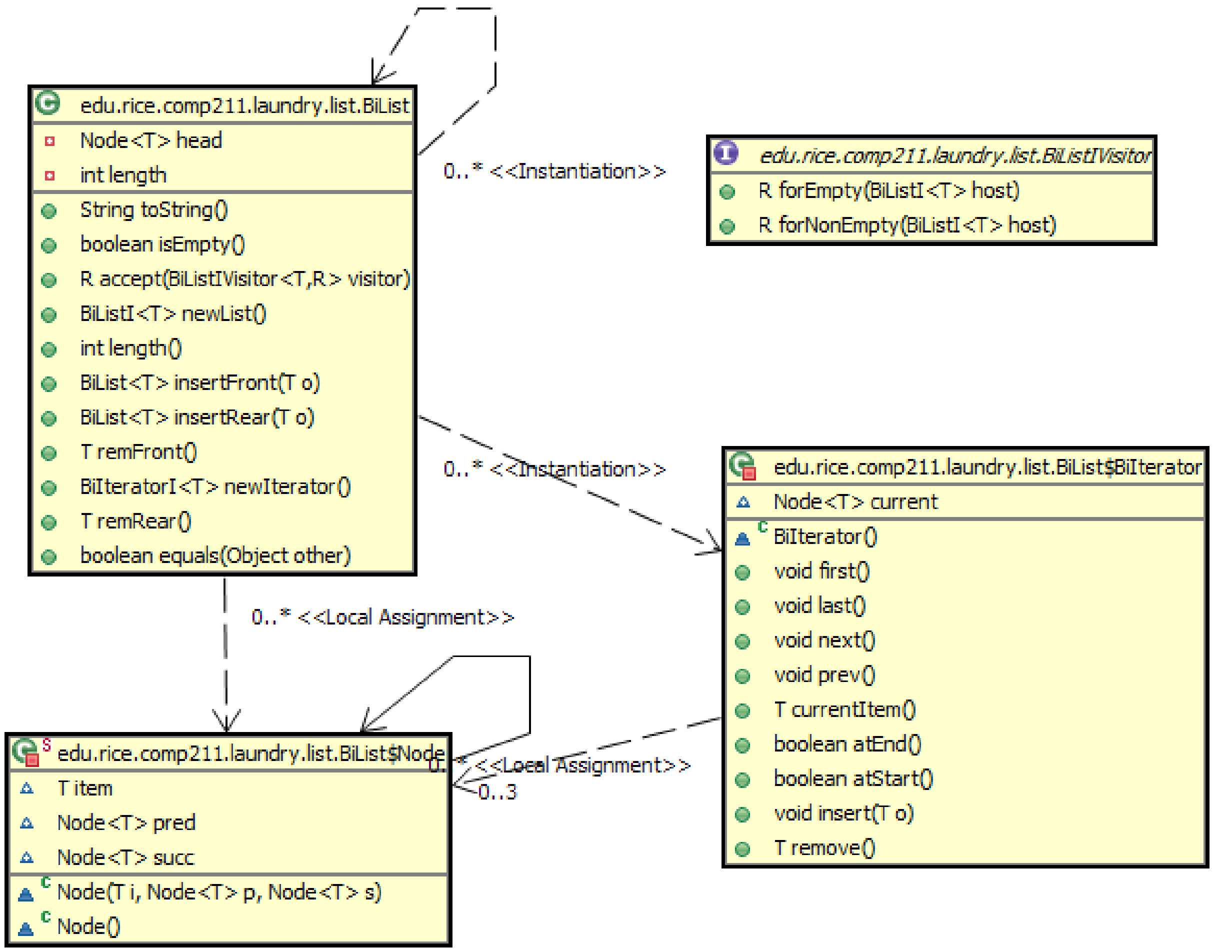


- Fast access to the end of the list
- More complicated. More “heavy weight”. Possibly slower.
- No terminating class – must use loops with iterators

BiList structure



The mutating Nodes are hidden behind the "encapsulation barrier" of the BiList class.



```
edu.rice.comp211.laundry.list.BiList
  Node<T> head
  int length
  String toString()
  boolean isEmpty()
  R accept(BiListIVisitor<T,R> visitor)
  BiListI<T> newList()
  int length()
  BiList<T> insertFront(T o)
  BiList<T> insertRear(T o)
  T remFront()
  BiIteratorI<T> newIterator()
  T remRear()
  boolean equals(Object other)
```

```
edu.rice.comp211.laundry.list.BiListIVisitor
  R forEmpty(BiListI<T> host)
  R forNonEmpty(BiListI<T> host)
```

```
edu.rice.comp211.laundry.list.BiList$BiIterator
  Node<T> current
  BiIterator()
  void first()
  void last()
  void next()
  void prev()
  T currentItem()
  boolean atEnd()
  boolean atStart()
  void insert(T o)
  T remove()
```

```
edu.rice.comp211.laundry.list.BiList$Node
  T item
  Node<T> pred
  Node<T> succ
  Node(T i, Node<T> p, Node<T> s)
  Node()
```

BiList code snippets

Here are some code snippets that show you how the insertFront and insertRear operations are done in BiList:

```
public BiList<T> insertFront(T o) {
   Node<T> next = head.succ;
   Node<T> newNode = new Node<T>(o, head, next);  // allocate new Node
   // insert new Node
   head.succ = newNode;
   next.pred = newNode;
   length++;
   return this;
}

public BiList<T> insertRear(T o) {
   Node<T> prev = head.pred;
   Node<T> newNode = new Node<T>(o, prev, head);  // allocate new Node
   // insert new Node
   head.pred = newNode;
   prev.succ = newNode;
   length++;
   return this;
}
```

BiList code snippets, continued…

Here are some code snippets that show you how the remFront and remRear operations are done in BiList:

```
 public T remFront() {
      if (isEmpty()) {
         throw new IllegalStateException("Attempt to remove first element of empty BiList");
      }
      Node<T> next = head.succ;
      T remOb = next.item;
      Node<T> newNext = next.succ;
      head.succ = newNext;
      newNext.pred = head;
      length--;

      return remOb;
   }

   public T remRear() {
      if (isEmpty()) {
         throw new IllegalStateException("Attempt to remove last element of empty BiList");
      }
      Node<T> last = head.pred;
      T remOb = last.item;
      Node<T> lastPred = last.pred;
      head.pred = lastPred;
      lastPred.succ = head;
      length--;
      return remOb;
   }
```

Comments on BiList code

Supports the *iterator* design pattern, which is applicable to any data structure that holds a collection of items. An *iterator* for a collection enables the elements to be processed as a sequence in some order.

Key operations:

- *Factory method* (design pattern) for constructing an iterator
- Method for advancing the iterator cursor
- Method for getting the current item
- Method for testing whether cursor is at the end of enumerating the collection.

BiIterator class from BiList.java

```
private class BiIterator implements BiIteratorI<T> {
   Node<T> current;
   BiIterator() {
      current = BiList.this.head.succ;  // current is first item (if one exists)
   }
   public void first() {
      current = BiList.this.head.succ;  // current is first item (if one exists)
   }
   public void last() {
      current = BiList.this.head.pred;  // current is last item (if one exists)
   }
   public void next() {
        current = current.succ;         // wraps around end including header
   }
   public void prev() {
      current = current.pred;           // wraps around end including header
   }
   public T currentItem() {
      if (current == BiList.this.head) {
         throw new IteratorException("No current element in " + BiList.this);
      }
      return current.item;
   }
   public boolean atEnd() { return current == BiList.this.head; }
}
```

BiIterator class commentary

`BiIterator` is an advanced Java class. Why?

- It is an inner class.
- What is the difference between static and dynamic inner classes
- It is `private`
- It is generic.
- Why isn't it declared as `BiIterator<T>`?

What is the scope of a type parameter?

- It is private.

While Loops

A "while" loop is a program structure that executes a set of Java statements while a particular condition is "true" (remember: "*do if true*").

The basic Java syntax is:

```
while(boolean_expression) {
   // set of statements
}
```

The boolean expression is **re-evaluated every time** the loop repeats, until it evaluates to "false", at which point the loop **immediately** terminates and the program resumes at the line right after the loop.

Using an Iterator

"`!iterator.atEnd()`" is usually the best boolean expression to use. Note the use of the boolean negation because we want "not at the end".

```
public static <T> boolean delete(BiListI<T> host, T elt) {
  IteratorI<T> it = host.newIterator()

  while(!it.atEnd()) {
     if (it.currentItem().equals(elt)) {
            it.remove();
            return true;
         }
     it.next()
  }
  return false;
}
```

Delegation Model Programming

- **"If your code depends on what an object is (i.e. its class type), then delegate to it."**
  - Use the fact that an object knows what to do –polymorphism.
- Minimize the use of conditionals
  - Delegate to the object (call a method on the object) rather than extracting information from the object and using a conditional to process that information.
- Techniques:
  - Interpreter pattern: put methods on each type of object that enables it to do it's own processing → polymorphic dispatching to that object.
  - Visitor pattern: Code type-dependent processes as cases of a visitor. Do the processing of the unknown object by having it accept the visitor.

Note: Using iterators is generally not considered consistent with delegation model programming. Real program generally use a mix of imperative/procedural styles like iterators mixed with delegation styles like recursion, polymorphic dispatching, etc.