Mutable Linked Lists

Corky Cartwright

Department of Computer Science

Rice University

Background

- 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`**. These fields are the addresses of:
  - the node (chunk of memory) representing the first element in the list rooted at **`N`** and
  - the **`Cons`** node **`N'`** representing the list (**`rest`**) 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 by assignment statements executed *after* initialization.
- Mutation can be performed with discipline and taste. We will focus initially on the mutable generalization of composite lists.

Pure Mutable Generalization of Functional Lists

- In the notes OO Design, I introduce the notion of Quasi-functional Lists (LRS structures in the terminology of Nguyen and Wong) which generalizes the composite formulation of functional lists by making the **`first`** and **`rest`** fields *mutable*.
- But Quasi-functional lists provide no asymptotic speed-up over functional lists. Inserting or removing elements from the end of a list takes O(*n*) time.
- Traditional linked lists can provide asymptotic speed-ups.
- Disciplined use of mutation
  - Never modify fields directly.
  - Support high level mutation via mutating methods.

Example: BiLists

- In the notes on OO design, I introduce traditional mutable *singly-linked* lists before discussing *doubly-linked* lists. As we hae seen functional lists are singly linked. Mutable *singly-linked* lists are lighter weight (simpler and, in many cases, faster) than mutable *doubly-linked* lists. Allowing mutation on singly linked lists can asymptotically speed-up some operations on lists, but others (such as deleting the last element of a list) take O(n) time in the absence of double-linking.
- Furthermore, formulating nodes as objects adds weight (a two word header in each node) to a linked-list implementation so double-linking adds only modest extra space (one word) and time cost more space takes more time).
- A doubly-linked representation adds a predecessor address field to each Cons node.

Comments BiList code

- Discussed in detail in OO Design notes.
- Supports the *iterator* design pattern, which is applicable to any data structure that holds a collection of items.
- Key operations involved in the *iterator* pattern:
  - Factory method for constructing an iterator (in collection class)
  - Method for advancing the iterator cursor (in iterator interface)
  - Method for getting the current item (in iterator interface)
  - Method for testing whether cursor is at the end enumerating the collection (in iterator interface).

For Next Class

- New homework due next Wednesday. Assignmen specs are much longer than the code you must write. Straightforward but not conducive to last-minute solution.
- Two forms for supporting code base:
  - Class per file (prepares you for last two assignments)
  - All classes in one file (easier)
- DrJava makes it easy to practice writing code fragments/exercises. Do it! Don't be afraid to experiment. The interactions pane makes it easy.