Advanced Programming

List Implementations
That Link Data
Topics

- Linked Data
  - Forming a Chains
- The Class **Node**
- A Linked Implementation
  - Adding to End of List
  - Adding at Given Position
  - Method `remove`
  - Method `replace`
  - Method `getEntry`
- Method `contains`
- Remaining methods
- Using a Class Node with Set and Get Methods
- Tail References
  - Revision of List
- Pros and Cons of Using Chain
- Java Class Library: LinkedList
Linked Data

Consider the analogy of desks in a classroom

- Placed in classroom as needed
- Each desk has a unique id, the “address”
- The desks are linked by keeping the address of another chair
- We have a chain of chairs
Linked Data

Fig. 6-1 A chain of 5 desks.
Forming a Chain

- First desk placed in room
  - Blank desk top, no links
  - Address of the first chair given to teacher

Fig. 6-2 One desk in the room.
Forming a Chain

- Second student arrives, takes a desk
  - Address of first desk placed on new desk
  - Instructor “remembers” address of new desk

Fig. 6-3 Two linked desks
Forming a Chain

- Third desk arrives
  - New desk gets address of second desk
  - Instructor remembers address of new desk

Fig. 6-4 Three linked desks, newest desk first.
Forming Another Chain

- This time the first student is always at the beginning of the chain
  - Instructor only remembers first address
- The address of a new desk is placed on the previous desk (at end of chain)
  - End of chain found by following links
  - Newest desk does not have a pointer address to any other desk
Forming Another Chain

Fig. 6-5 Two linked desks, newest desk last.
Forming Another Chain

Fig. 6-6 Three linked desks, newest desk last.
Forming Another Chain

Fig. 6-7 Five linked desks, newest desk last.
Forming Yet Another Chain

- Consider the requirement to organize the chain alphabetically
  - New arrivals are placed somewhere in the chain, not necessarily at the end

- Possibilities for placement of a new desk
  - Before all current desks
  - Between two existing desks
  - After all current desks
Forming Yet Another Chain

Fig. 6-8  Chain of desks prior to adding a new desk to beginning of the chain
Forming Yet Another Chain

Fig. 6-9  Addition of a new desk to beginning of a chain of desks
Forming Yet Another Chain

Fig. 6-10  Two consecutive desks within a chain prior to adding new desk between

newDesk

12

deskBefore

20

deskAfter

5

8
Forming Yet Another Chain

Fig. 6-11  Addition of a new desk between two other desks.
The Class **Node**

- Nodes are objects that are linked together to form a data structure.

- We will use nodes with two data fields:
  - A reference to an entry in the list
    - (the person sitting at the desk)
  - A reference to another node
    - (the address on the paper on the desk)

- [View class definition](#) of Node
private class Node {

    private T data; // entry in list
    private Node next; // link to next node

    private Node(T dataPortion) {
        data = dataPortion;
        next = null;
    } // end constructor

    private Node(T dataPortion, Node nextNode) {
        data = dataPortion;
        next = nextNode;
    } // end constructor

} // end Node
The Class Node

Fig. 6-17 Two linked nodes with (a) primitive data; (b) object data
A Linked Implementation of the ADT List

- Use a chain of nodes
  - Remember the address of the first node in the chain
- Record a reference to the first node
  - The “head reference”
- The implementation contains the class Node as an inner class
public class LList<T> implements ListInterface<T> {
    private Node firstNode; // reference to first node
    private int length; // number of entries in list
    public LList() {
        clear();
    } // end default constructor
    public final void clear() {
        firstNode = null;
        length = 0;
    } // end clear
    /* < Implementations of the public methods add, remove, replace, getEntry, contains, getLength, isEmpty, isFull, and display go here. >
    ... */
// ---------------private!-----------------------------
/** Task: Returns a reference to the node at given position.
 * Precondition: List is not empty; 1 <= givenPosition <= length.
 */
private Node getNodeAt(int givenPosition) {
    // < Implementation deferred >
} // end getNodeAt

private class Node // private inner class
{
    // < See few slides before>
} // end Node
} // end LList
Choosing a Core Group of Methods

- Must be able to create the collection
  - Need method `add()`
- Desire to see objects that have been added to the list
- Need method `display()`
  - Constructor also necessary
Adding to the End of the List

Fig. 6-18 (a) An empty list and a new node; (b) after adding a new node to a list that was empty
Fig. 6-19 A chain of nodes (a) just prior to adding a node at the end; (b) just after adding a node at the end.
Adding at a Given Position Within the List

Fig. 6-20 A chain of nodes (a) prior to adding a node at the beginning; (b) after adding a node at the beginning.
AddMethods

public boolean add(T newEntry)
public boolean add(int newPosition, T newEntry)

2 cases:
  1. Adding entry to beginning of list
  2. Adding entry to some other position
Adding at a Given Position Within the List

Fig. 6-21 A chain of nodes (a) prior to adding node between adjacent nodes; (b) after adding node between adjacent nodes
Other Core Methods Implemented

- Method `isEmpty()`
  - Note use of `assert` statement
- Method `display()`
  - Sends output to `System.out`
- Private method `getNodeAt()`
Testing the Incomplete Implementation

- Core group of methods implemented
  - Now to be tested
  - View test program and output
- Other methods specified as stubs to satisfy syntax checker

```java
public T remove (int givenPosition)
{
    return null;
} // end remove
```
Pros and Cons of a Chain for an ADT List

- The chain (list) can grow as large as necessary
- Can add and remove nodes without shifting existing entries

But …

- Must traverse a chain to determine where to make addition/deletion
- Retrieving an entry requires traversal
  - As opposed to direct access in an array
Literature

• Chapter 6
  • *Data Structures and Abstractions with Java, by Carrano, Second Edition.*