A List Implementation That Links Data

Chapter 14

Javadoc: http://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html

Pre-requisite: Chapters 3, 4, 12
Objectives

- Describe linked organization of data
- Implement *add* methods of the ADT list by
  - using linked chain of nodes
- Test partially complete implementation of a class
Contents

• Operations on a Chain of Linked Nodes
  ▪ Adding a Node at Various Positions
  ▪ Removing a Node from Various Positions
  ▪ The Private Method getNodeAt

• Beginning the Implementation
  ▪ The Data Fields and Constructor
  ▪ Adding to the End of the List, or at a Given Position Within the List
  ▪ The Methods isEmpty and toArray
  ▪ Testing the Core Methods

• A Refined Implementation
  ▪ The Tail Reference

• The Efficiency of Using a Chain to Implement the ADT List

• Java Class Library: The Class LinkedList
The private class **Node**

- Figure 3-5 Two linked nodes that each reference object data

![Diagram of two linked nodes](image)

- Recursive class definition (similar to Scheme list in Csci 1901)

```java
class Node
{
    private T data; // entry in bag
    private Node next; // link to next node

    < Constructors >
    . . .

    < Accessor and mutator methods: getData, setData, getNextNode, setNextNode >
    . . .

} // end Node
```
private class Node {
    private T data;
    private Node next;
    private Node(T dataPortion) {
        this(dataPortion, null);
    }
    private Node(T dataPortion, Node nextNode) {
        data = dataPortion;
        next = nextNode;
    }
    // ...
private class Node {
    private T data;
    private Node next;
    private Node(T data portion) { this(dataPortion, null); }
    private Node(T dPart, Node nNode) { data = dPart; next = nNode; }
    private T getData() { return data; }
    private void setData(T newData) { data = newData; }
    private Node getNextNode() { return next; }
    private void setNextNode(Node nextNode) { next = nextNode; }
}

Exercise: Match methods in Node class with following Scheme functions:
(a.) car (b.) cdr (c.) cons (d.) set! (e.) set-car! (f.) set-cdr! (g.) list
Terminology with Linked Nodes

List
- Empty List
- Non-empty List

Position (in non-empty List)
- First
- Middle
- Last
Recall List ADT (Listing 12-1)

```java
public interface ListInterface<T> {
    public void add(T newEntry);
    public void add(int newPosition, T newEntry);
    public T remove(int givenPosition);
    public void clear();
    public boolean replace(int givenPosition, T newEntry);
    public T getEntry(int givenPosition);
    public boolean contains(T anEntry);
    public int getLength();
    public boolean isEmpty();
    public T[] toArray();
}
```

// Q? Compare with java.util.List interface using java-doc, e.g.,


- Positions inside a list start with a 0
- Method get(int index) instead of getEntry(int givenPosition), size() instead of getLength()
- Many more methods, e.g., equals, indexOf, iterator(), listIterator(),
  ```java
  • Remove(Object O), removeAll(Object O), retainAll(Collection c), subList(int fromIndex, toIndex)
  ```

Old Java version uses Object class instead of generic type
public class LList<T> implements ListInterface<T> {

    private class Node {
        private T data;
        private Node next;
        private Node(T data portion) { this(dataPortion, null); }
        private Node(T dPart, Node nNode) { data = dPart; next = nNode; }
        private T getData() { return data; }
        private void setData(T newData) { data = newData; }
        private Node getNextNode() { return next; }
        private void setNextNode(Node nextNode) { next = nextNode; }
    }

    private Node firstNode;
    private int numberOfEntries;

    public Llist() { clear(); }
    public final void clear() { firstNode = null; numberOfEntries = 0; }

    ....

} // Note: Java class java.util.LinkedList<E> is similar, but
// implements more interfaces, e.g., Iterable<E>, Collection<E>, Deque<E>, List<E>, Queue<E>
// has more methods, e.g., addFirst(), addLast(), getFirst(), getLast(), pop(), push(), ...
// See http://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html
/** Adds a new entry to this bag. 
 * @param newEntry the object to be added as a new entry 
 * @return true */

public boolean add(T newEntry) { // OutOfMemoryError possible

    // add to beginning of chain:
    Node newNode = new Node(newEntry);
    newNode.next = firstNode; // make new node reference rest of chain
                                // (firstNode is null if chain is empty)
    firstNode = newNode; // new node is at beginning of chain
    numberOfEntries++;

    return true;
} // end add
Add(T newEntry)

**Exercise:** Compare following 2 implementations of `add(T new Entry)` on

(a.) on complexity and list structure
(b.) closeness to `java.util.LinkedList.add(T e)` method semantics

(http://docs.oracle.com/javase/6/docs/api/java/util/LinkedList.html)

// Implementation A on pp. 349 in chapter 14
public void add(T newEntry) {
    Node newNode = new Node(newEntry);
    if (isEmpty())   firstNode = newNode;
    else  { lastNode = getNodeAt(numberOfEntries);
            lastNode.setNextNode( newNode );
    }
    numberOfEntries++ ;
}

// Implementation B from chapter 3
public void add (T newEntry) {
    Node newNode = new Node(newEntry);
    newNode.next = firstNode;
    firstNode = newNode;
    numberOfEntries++ ;
}
add(int newPosition, T newEntry)

• Ex. Review code on pp. 350 in chapter 14 for following case:
  ▪ Case 1: Chain is empty
  ▪ Case 2: Adding node at chain’s beginning
  ▪ Case 3: Adding node between adjacent nodes
  ▪ Case 4: Adding node to chain’s end
Case 1: Empty Chain

Node newNode = new Node(newEntry);
firstNode = newNode;

Figure 14-1 (a) An empty chain and a new node; (b) after adding the new node to a chain that was empty
Case 2: beginning

Node newNode = new Node(newEntry);
newNode.setNextNode(firstNode);
firstNode = newNode;

(a) firstNode
newNode

(b) firstNode
newNode

Figure 14-2 A chain of nodes (a) just prior to adding a node at the beginning; (b) just after adding a node at the beginning
Figure 14-3 A chain of nodes (a) just prior to adding a node between two adjacent nodes; (b) just after adding a node between two adjacent nodes

```java
Node newNode = new Node(newEntry);
Node nodeBefore = getNodeAt(newPosition - 1);
Node nodeAfter = nodeBefore.getNextNode();
newNode.setNextNode(nodeAfter);
nodeBefore.setNextNode(newNode);
```
Case 4: End

FIGURE 14-4 A chain of nodes
(a) prior to adding a node at the end;
(b) after locating its last node;
(c) after adding a node at the end

Node newNode = new Node(newEntry);
Node lastNode = getNodeAt(numberOfNodes);
lastNode.setNextNode(newNode);
Removing a Node from Various Positions

- Case 1: Removing first node

```java
firstNode = firstNode.getNextNode();
```

![Diagram](image.png)

**FIGURE 14-5** A chain of nodes (a) just prior to removing the first node; (b) just after removing the first node
Removing a Node from Various Positions

• Case 2: Removing node other than first

Node nodeBefore = getNodeAt(givenPosition - 1);
Node nodeToRemove = nodeBefore.getNextNode();
Node nodeAfter = nodeToRemove.getNextNode();
nodeBefore.setNextNode(nodeAfter);
nodeToRemove = null;

FIGURE 14-6 A chain of nodes (a) just prior to removing an interior node; (b) just after removing an interior node
The Private Method `getNodeAt`

- Returns reference to node at specified position

```java
private Node getNodeAt(int givenPosition)
{
    assert (firstNode != null) &&
    (1 <= givenPosition) && (givenPosition <= numberOfNodes);
    Node currentNode = firstNode;

    // traverse the chain to locate the desired node
    for (int counter = 1; counter < givenPosition; counter++)
    {
        currentNode = currentNode.getNextNode();
    }

    assert currentNode != null;

    return currentNode;
}
```

- **Ex.** Compare with Array implementation of `getNodeAt()`
  - computational complexity
Beginning the Implementation

• Design Decision:
  ▪ The structure of the chain of linked nodes

• Add statements at beginning
  ▪ Use code from Cases 1 and 2

• Add at end
  ▪ Must invoke `getNodeAt`
  ▪ Could maintain both head and tail references (deferred until later)
Figure 14-7 A linked chain with (a) a head reference; (b) both a head reference and a tail reference
Implementation

- View whole class, **Listing 14-1**
- Note
  - Methods for **add**
  - Method **isEmpty**
  - Method **toArray**
- Consider initial test program, **Listing 14-2**
  - **Output** of listing 14-2
Implementation

• Finishing the implementation
  ▪ Method `remove`
  ▪ Method `replace`
  ▪ Method `contains`

• Refining the implementation
  ▪ Add tail reference
  ▪ Avoids traversal of entire chain when `add` is called
Figure 14-8 A linked chain with both a head reference and a tail reference
Implement with Tail Reference

• Must alter method clear
  - `lastNode = null;`
• Adding to end of list
  - For empty list, head and tail reference new node
  - For non-empty list, use `lastNode.setNextNode(newNode);`
  - `lastNode = newNode;`
• View changes, **Listing 14-A**
FIGURE 14-9 Adding a node to the end of a nonempty chain that has a tail reference

After executing
lastNode.setNextNode(newNode);

After executing
lastNode = newNode;

Copyright ©2012 by Pearson Education, Inc. All rights reserved
Ex. Compare two implementations (e.g., array based, linked) of ListInterface for storage and time complexity.

**Storage growth:** smoother for linked implementation

**Accessing random positions:** faster in array implementation

---

<table>
<thead>
<tr>
<th>Operation</th>
<th>AList</th>
<th>LList</th>
<th>LList2</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(newEntry)</td>
<td>O(1)</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>add(newPosition, newEntry)</td>
<td>O(n); O(1)</td>
<td>O(1); O(n)</td>
<td>O(1); O(n); O(1)</td>
</tr>
<tr>
<td>toArray()</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>remove(givenPosition)</td>
<td>O(n); O(1)</td>
<td>O(1); O(n)</td>
<td>O(1); O(n)</td>
</tr>
<tr>
<td>replace(givenPosition, newEntry)</td>
<td>O(1)</td>
<td>O(1); O(n)</td>
<td>O(1); O(n); O(1)</td>
</tr>
<tr>
<td>getEntry(givenPosition)</td>
<td>O(1)</td>
<td>O(1); O(n)</td>
<td>O(1); O(n); O(1)</td>
</tr>
<tr>
<td>contains(anEntry)</td>
<td>O(n)</td>
<td>O(n)</td>
<td>O(n)</td>
</tr>
<tr>
<td>clear(), getLength(), isEmpty()</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

**Figure 14-11** The time efficiencies of the ADT list operations for three implementations, expressed in Big Oh notation
Design Decisions

• Efficiency of execution vs. implementation time

• Issues include:
  ▪ Access time
    • Add, remove, search
  ▪ Memory usage
    • Overhead for pointers
    • Wasted memory for arrays
Java Class Library: The Class LinkedList

• **ListInterface** similar to what we have defined
  ▪ Has more methods
  ▪ May use different name for method

• **Class LinkedList**
  ▪ Implements List, Queue, Deque
End

Chapter 14
/** Retrieves all entries that are in this bag. 
* @return a newly allocated array of all the entries in the bag */

public T[] toArray()
{
    // the cast is safe because the new array contains null entries
    @SupressWarnings("unchecked")
    T[] result = (T[]) new Object[numberOfEntries]; // unchecked cast
    int index = 0;
    Node currentNode = firstNode;
    
    while ((index < numberOfEntries) && (currentNode != null))
    {
        result[index] = currentNode.data;
        index++;
        currentNode = currentNode.next;
    } // end while

    return result;
} // end toArray