Linked Nodes

How I spent my youth as a CS graduate student

Reading LDC Ch 14.10 - 14.14
Linking Objects

Arrays are limited in the sense they have a fixed size

- Resizing as needed must be done carefully and is not efficient

A *structure of linked objects* is the primary *alternative* to an array-based implementation of a collection

**Initial** implementation idea: a *Person* as part of a community

```java
public class Person{
    private String name;
    private String address;
    private Person next;
    // and whatever else
}
```
An OK Idea: Objects with Links

Using only this one class, a linked structure is created

- One Person object contains a link to another Person object
- This second object contains a reference to a third Person, etc.
- This type of object is sometimes called self-referential
- This kind of relationship forms the basis of a linked list

- Has capacity limited only by memory in the computer
- Is a dynamic structure because its size grows and shrinks as needed to accommodate the number of objects stored
A flaw in the Person class is that the self-referential Person class must be designed so that it “knows” it may become a node in a linked list. This violates the goal of separating the implementation details from the parts of the system that use the collection.

We better separate the details of the linked list structure from the objects that the list stores: Define a separate node class LinearNode that serves to link the objects together.
Managing Linked Lists

is more complicated than managing an array

A node may be inserted or deleted at any location
- at the front of the list,
- among the interior nodes, or
- at the end of the list

There are a few basic techniques when managing nodes on the list, no matter what the list is used to store

Special care must be taken when dealing with the first or last node in the list so that the reference to the entire list is maintained appropriately
Inserting a node at the front
Inserting a node in the middle
Deleting the first node in the list
Deleting an interior node

front

previous  \( \textbf{1} \)  current

\( \textbf{2} \)
package javafoundations;

public class LinearNode<T> {
    private LinearNode<T> next;
    private T element;
    //Constructor: Creates an empty node
    public LinearNode() {
        next = null;
        element = null;
    }
    //Constructor: Creates node with element
    public LinearNode(T elem) {
    }
}
/**@return the node that follows this one */
public LinearNode<T> getNext()
{
}

/** Sets the node that follows this one */
public void setNext (LinearNode<T> node)
{
}

/** @return element stored in this node */
public T getElement()
{
}

/** Sets the element stored in this node */
public void setElement (T elem)
{
}
LinkedStack:

Stack Implementation with Linear Nodes
package javafoundations;

public interface Stack<E> {

    /** @param newItem is added to the top of the stack. */
    public void push(E newItem) {

    /** @return the removed top element from the stack. */
    public E pop();

    /** @return the top element of this stack without removing it. */
    public E peek();

    /** @return true iff the stack contains no elements. */
    public boolean isEmpty();

    /** @return the number of elements in the stack. */
    public int size();

    /** @return a string representation of the stack. */
    public String toString();
}
Stack Animation

Adding an element

Removing an element

Top of stack
/** Represents a linked implementation of a stack. */
package javafoundations;

import javafoundations.exceptions.*;

public class LinkedStack<T> implements Stack<T> {
    private int count;
    private LinearNode<T> top;

    // Constructor: Creates an empty stack.
    public LinkedStack() {
        count = 0;
        top = null;
    }

    (more...)}
What a LinkedStack looks like
/** @return the top element of the stack */
public T peek () throws EmptyCollectionException {
    if (count == 0) {
        throw new EmptyCollectionException
            ("Peek failed. Stack is empty.");
    }
}

/** @return true IFF stack is empty */
public boolean isEmpty() {
}

/** @return the number of elements on the stack */
public int size() {
}
Popping a node off a Stack

(we just need the element, though, not the node)
/** Removes the element at the top of this stack
 * @return a reference to it.
 * @throws an EmptyCollectionException if stack empty
 */
public T pop() throws EmptyCollectionException {
    if (count == 0)
        throw new EmptyCollectionException
            ("Pop failed. Stack is empty.");
}

(more...)
Pushing a node on to Stack
(slightly different than code in next slide)
/** Adds the element at the top of this stack  
 * @param element will be “pushed” onto the Stack. */

public void push(T element) {

    // We need a LinearNode to point to element!
    //

    LinearNode<T> node = new LinearNode<T>();
    node.setElement(element);
    node.setNext(top);
    top = node;
    count++;
}

(more...)
/** @return a string representation of this stack. */
public String toString() {
    String result = "<top of stack>\n";
    LinearNode current = top;

    while (current != null) {
        result += current.getElement() + "\n";
        current = current.getNext();
    }
    return result + "<bottom of stack>";
}
java.util.LinkedList<E>

NOTICE:
This is the official Java implementation,
Doubly Linked Lists

- A simple linked list is only one kind of linked structure
- In a *doubly linked list*, each node in the list stores both a reference to the next element and a reference to the previous one
- Java’s library implementation uses doubly linked lists
Java’s LinkedList\(<E>\) “feels” like both an array and a linked list:

Even though Java uses doubly linked list, we will use the picture of a single-linked list because it is simpler and conveys the same idea.

To use it you need to start your code by

```java
import java.util.LinkedList;
```

To get a new list:

```java
LinkedList\(<E>\) L = new LinkedList\(<E>() ;
```
Methods for Adding LinkedList Nodes

```java
public void add (int index, E element)
    indices change automatically!
    if index > size() throws IndexOutOfBoundsException

public void addFirst (E element)
public void addLast (E element)
aka: public void void add (E element)

LinkedList<String> L1 =
    new LinkedList<String>();
L1.add(0, "eggs");
L1.addFirst("milk");
L1.addLast("bread");
L1.add(2, "chicken");

System.out.println("contents of L1: " + L1);
```
Methods for Getting List Items

```java
public E getFirst ()

public E getLast ()

public E get (int index)

LinkedList<String> L2 = new LinkedList<String>();
for (int i = L1.size() - 1; i >= 0; i--)
    L2.add(L1.get(i));
System.out.println("contents of L2: " + L2);
```

What does L2 contain?
Methods for Setting and Removing Items

public E set (int index, E element)

public E remove (int index)

public E removeFirst ()

public E removeLast ()

L1.remove(1);
L1.set(2, "beer");
L1.set(1, L1.removeFirst());
L1.addFirst(L1.getLast());
L1.add(1, "butter");
System.out.println("new contents of L1: " + L1);

new contents of L1: (what happened to the beer?)
BTW: Java’s Stack<E> implementation is using Java’s LinkedList <E>

```java
public void push(E item)

public E pop() throws NoSuchElementException

public E element () // same as peek
```

![Stack representation](image)