CSC 102 Lecture Notes Week 8
More on Linked Lists and Iterators
Introduction to Recursion
More on Searching and Sorting

I. **Relevant reading.**

A. Chapter 13 -- Recursion

B. Chapter 14 -- Sorting and Searching (again)

C. Chapter 15 -- Data Structures (again)

II. **Announcemnents**

A. Program 6 will be released on Friday 24 May.

B. The 100% Program 5 due date is Friday 24 May.

C. Any lingering Program 4 demos will be done this week in lab.

III. **Linked List and Iterator Implementations**

A. There is a very good example of linked list implementation from chapter 15 of book.

B. It’s online in 102/examples/book/ch15/impllist/

C. It shows the implementation of methods for a singly-linked non-generic list.

D. It also shows implementation of an iterator for this structure.

E. We’ll walk through the example in detail during lecture on Monday.

F. It’s directly relevant to the work you’ll do in Program 6.

IV. **Midterm Review**

A. On Wednesday of this week, we’ll walk through the full solution to the midterm, and discuss.

V. **Introduction to recursion (Ch 13).**

A. It’s a useful problem solving technique.

B. It makes some problem solutions much easier, in particular problems that in involve accessing linked data.

C. It needs to be part of a programmer’s "tool bag".

VI. **The fundamental idea of recursion.**

A. Subdivide a large problem into separate parts.

B. Solve one simple part of the problem.

C. Apply the solution to the rest of the problem.

VII. **A simple searching example**

A. Suppose we want a list `elementOf` method.

1. it returns true if a list contains a particular element

2. it returns false if not

B. An *iterative* solution uses a familiar for loop, e.g.,

```
import java.util.*;

/****
 *
 * This class illustrates an iterative  elementOf method.  Compare it to the
 * recursive solution in ./RecursiveElementOf.java.
 */
public class IterativeElementOf {

    /**
     * Return true if the given element is in the given list, false if not.
     * The method uses a standard form of for loop to examine each element of
     * the list, returning true if we find the element we’re looking for, false
     * if we run off the end of the list.
     */
    static <E> boolean elementOf(List<E> list, E element) {

        for (int i = 0; i < list.size(); i++) {
            if (element.equals(list.get(i))) {
                return true;
            }
        }
        return false;
    }
}
```

C. A *recursive* solution goes like this:

1. If the list is empty, return false.
2. If the element is first in the list, return true.
3. Otherwise, search the rest of the list ***recursively.***

D. Here’s the code, which does the same work as the preceding iterative version:

```
import java.util.*;

/****
 *
 * This class illustrates a recursive elementOf method.  Compare it to the
 * iterative solution in ./IterativeElementOf.java.
 *
 */
public class RecursiveElementOf {

    /**
     * Return true if the given element is in the given list, false if not.
     * The method uses a recursive search algorithm, consisting of the
     * following three steps:
     *
     *    (1) If the list is empty, return false.
     *
     *    (2) If the element we’re looking for is the first in the list,
     *        return true.
     *
     *    (3) Otherwise, search for the element recursively in the rest of the
     *        list, i.e, the sublist from the second through the last element.
     */
    <E> boolean elementOf(List<E> list, E element) {
        /*
         * Step 1.
         */
        if (list.size() == 0) {
            return false;
        }

        /*
         * Step 2.
         */
        if (list.get(0).equals(element)) {
            return true;
        }

        /*
         * Step 3.
         */
        return elementOf(list.subList(1, list.size()), element);

    }

}
```

VIII. **Analysis of `elementOf` performance.**

A. Here's a summary of the performance for the iterative and recursive solutions to `elementOf` on an ArrayList and a LinkedList:

1. The iterative solution works well on an ArrayList, in O(n) time.
2. The iterative solution works poorly on a LinkedList, in O(n^2) time.
3. The recursive solution works well an ArrayList, in O(n) time.
4. The recursive solution works well on a LinkedList, in O(n) time.

B. ***Question:*** how do you explain these behaviors?

. . . thinking . . .

C. OK, so you've thought about it for a couple seconds; have a look at Figure 1, and consider the following points:

1. The `get(int)` operation is constant on `ArrayList`s, whereas it's linear on `LinkedList`s.
2. The `getFirst()` and `getRest()` methods are both constant on `LinkedList`s. (These methods are equivalent to, respectively, get(0) and subList(1,list.size()).)
3. The `subList` method is constant on both ArrayLists and LinkedLists, due to Java's implementation of java.util.AbstractList. (Note that if ArrayList.subList created a new array, it would be O(n). Think about what might be going on inside the ArrayList implementation to make subList O(1).)
4. Think (some more) about what these observations mean for the performance of the iterative versus recursive searches on `ArrayList` versus `LinkedList`.