Iterators

See Weiss p. 240-242, 574-579

If you have data stored in an ArrayList, it is an easy matter to go to the nth element of the list -- because arrays are contiguous blocks of memory, the address of the nth element can be computed as an offset from the start of the array.

The only way to find the nth element of a linked list is to start at the beginning and walk n steps.

Suppose you want to sum the first n elements of a list L of integers:

```
int sum = 0;
for (int i = 0; i < n; i++)
        sum += L.get(i);
```

For ArrayLists each get( ) is constant time, so this sum takes $O(n)$ time to compute, as you would expect. For LinkedLists get(i) takes i steps, so this sum takes time $O(n^2)$ to compute. This is terrible.

Of course, if you have access to the Node structure of the linked list you can find the sum in time O(n):

```
int sum = 0;
Node p = L.head;
while (p.next != null) {
        p = p.next;
        sum += p.data;
}
```

But one of the goals of object-oriented programming is to hide implementation details, so we don't want to give application programmers access to the Node structure.

Iterators are a solution to this problem. An iterator is an object that has a sense of its current location in the list and can move around efficiently. Here is code that works efficiently with both ArrayLists and LinkedLists:

```
Iterator it = L.iterator();
int sum = 0;
while (it.hasNext())
        sum += it.next();
```

There are two kinds of iterators for a list with base-type E. The basic Iterator interface is very simple, with just three methods:

- boolean hasNext( ), which tells you if there are more elements of the list
- E next( ), which gets the data value stored in the next element of the list
- void remove( ), which removes the element returned by the last call to next( ).

The ListIterator interface adds more methods to this class. For one thing, it allows you to move in either direction: it.next( ) and it.previous( ). There is also a set(e) method that changes the data in the last node that was referenced by next( ) or previous( ), and an add(e) method that inserts data into the list.

In Lab 4 you will implement a ListIterator for a doubly-linked list.

When you are thinking about what the iterator will do, it helps to think of its current position as halfway between two nodes: the nodes it will get to on calls to next( ) and previous( ).

```
┌─────┐     ┌─────┐     ┌─────┐     ┌─────┐     ┌─────┐
│     │ ──> │  7  │ ──> │ 11  │ ──> │ 13  │ ──> │     │
│     │ <── │     │ <── │     │ <── │     │ <── │     │
└─────┘     └─────┘     └─────┘     └─────┘     └─────┘
 head                                              tail

              current position
```
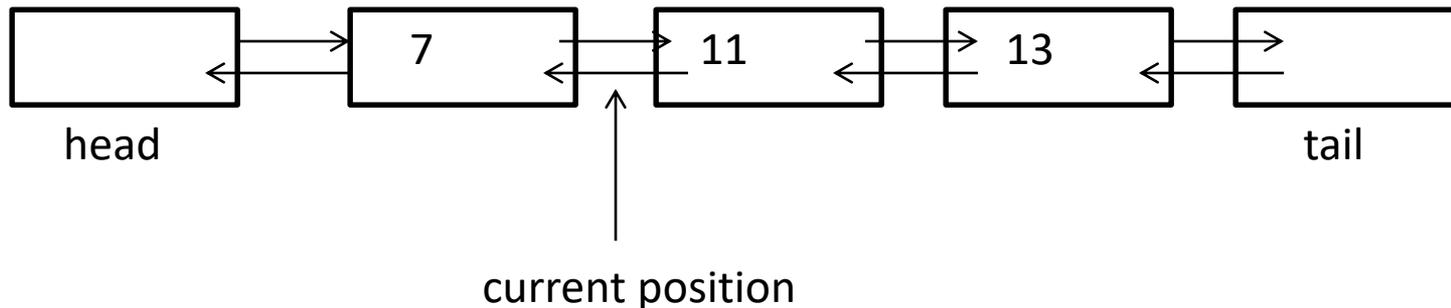
Your code will probably need to put the current marker at one node or another, but you can use this picture as a guide to behavior.

There are two issues you must face when implementing iterators. One is that some operations aren't allowed after operations that change the structure of the list. You can't do a set( ) operation immediately after an add( ) or remove( ). set( ) remove( ) and add( ) all assume that the previous operation was next( ) or previous( ) and they work on the value that was returned by that operation. Your iterator needs to keep track of this.

The other tricky aspect to iterators is that changes to the list structure could come from multiple sources. For example, an iterator might be used to walk along the nodes of a list deleting those that contain prime numbers. This might be running simultaneously with other list operations. If the iterator sees that the next node has value 23 and goes to delete it, it is possible that this node has been deleted by something else. In this case the iterator will end up deleting the wrong node.

To avoid this iterators throw an exception if they try to change the list if anything external to them (a list operation or another iterator) has changed the list since they were created. It is okay for something to change the list before your iterator is created, but if the list is changed after the iterator is created then the iterator itself is no longer able to make any changes.

This is actually easy to enforce. Give the list class a variable called modCount (for "modification count") and the iterator class a variable called myModCount. Anything that changes the list structure (iterator or list operation) should increment the list's modCount. When the iterator is created it takes the list's modCount as the starting value for its own myModCount. If the iterator changes the list, it increments both the list's and its own counts. If the iterator ever sees that the list's count is different from its own, it knows that something else has changed the list. Isn't that clever??