CSC 20 Chapter 16: Linked Lists

One way of storing data is in a linked structure. This has the benefit over arrays of not needing to know ahead of time how many elements will be stored. But, it is not random access and has higher memory overhead (because of the link storage).

```
public class ListNode {
    public int      data;
    public ListNode next;
}
```

If I wanted to store the number 1,2,3, I could manually do it this way.

```
ListNode first = new ListNode();
first.data = 1;
first.next = new ListNode();
first.next.data = 2;
first.next.next = new ListNode();
first.next.next.data = 3;
first.next.next.next = null;  // Indicate end of list
```

It's made simpler with a constructor in the `ListNode` definition.

```
public class ListNode {
    public int      data;
    public ListNode next;
    public ListNode(int data, ListNode next) {
        this.data = data;
        this.next = next;
    }
}
```

If I wanted to store the number 1,2,3, I could manually do it this way.

```
ListNode first = new ListNode(1,null);
first.next = new ListNode(2,null);
first.next.next = new ListNode(3,null);
```

Or, if you want to be super concise.

```
ListNode first = new ListNode(1,new ListNode(2,new ListNode(3,null)));
```

In class, we'll develop `MyLinkedList`; an implementation of a subset of `LinkedList<E>` functionality.

```
public class MyLinkedList<E> {

    private ListNode<E> first;
    private ListNode<E> last;
    private int len;

    public MyLinkedList() {
        first = null;
        last = null;
        len = 0;
    }

    public int size() {          // O(1) running time
        return len;
    }

    public void add(E d) {       // O(1) running time
        if (len==0) {
            first = new ListNode<E>(d,null);
            last = first;
        } else {
            last.next = new ListNode<E>(d,null);
            last = last.next;
        }
        len += 1;
    }

    public String toString() {
        if (len==0) {
            return "[]";
        } else {
            String res = "[" + first.data;
            ListNode<E> cur = first.next;
            while (cur != null) {
                res = res + "," + cur.data;
                cur = cur.next;
            }
            return res + "]";
        }
    }

    public static void main(String[] args) {
        MyLinkedList<Integer> l = new MyLinkedList<Integer>();
        System.out.println(l.size());
        System.out.println(l);
        l.add(10);
        System.out.println(l.size());
        System.out.println(l);
        l.add(20);
        System.out.println(l.size());
        System.out.println(l);
    }

}

class ListNode<E> {
    public E data;
    public ListNode<E> next;
    public ListNode(E data, ListNode<E> next) {
        this.data = data;
        this.next = next;
    }
}
```

[Do more in class]

Extra Features

Some people have `front` refer to a "dummy" node. A "dummy" node should not be considered part of the list, but is there to eliminate special code that would normally handle changes to `front`.

As written above, removal of an element from the back of the list is O(n) because it requires a full traversal of the list to update `last` after removal. By adding a `prev` field to `ListNode` and maintaining it to be a link to the previous element of the list, this can be reduced to O(1).

Last modified 4 March 2015