CS 151

Linked Lists and Iterators, continued
Prelab 4 due now.

Reminder of the course’s late policy: You are allowed to have 2 labs up to one day late. After that, up to 10% per day penalty is applied. When you DO use your late, let me know when you’ve handed it in so that I can tell the graders to re-run their fetch script. But there is no need to ask permission from me individually.
Recall that a linked list is a data structure that represents a sequence of elements that are stored non-contiguously in memory.

Supported Linked List Operations:
• void add( item, index ) – add an item at the specified index
• item remove( index ) – remove and return the item at the specified index
• int size() – return the number of elements in the stack
• boolean isEmpty() – return whether the stack is empty of any items
• void clear()/makeEmpty() – clear all items from the stack

We saw how to implement a linked list (using an inner Node class, that is, a Node class defined inside the LinkedList class) such that the size, isEmpty and makeEmpty methods are O(1), and the add and remove methods are O(n).
A **doubly-linked list** is a linked list where each node not only keeps track of the next element in the list, but also of the previous one.

The supported operations are the same as for the linked list, but our nodes have two pointers (next and previous), and our list itself has two entry points, front and back.

```java
class DoublyLinkedList<T> {
    class Node<T> {
        private T data
        private Node<T> next
        private Node<T> prev

        public Node(T data, Node<T> next, Node<T> prev) {
            this.data = data
            this.next = next
            this.prev = prev

        }...
    }
}
```
Doubly Linked List Implementation

On your prelab, you thought a bit about the add method:

```java
class DoublyLinkedList<T> {
    class Node<T> { ... }

    Node<T> front // points to the front of the list
    Node<T> back // points to the back of the list
    int size // the number of elements in the list

    void add( T item, int index ) {
        // What special cases must you test for?
        // Special case for size == 0?
        // Special case for index == 0?
        // Special case for index == size?
        // Special case for index == size-1?
    }
}
```
On your prelab, you thought a bit about the add method:

```java
public class DoublyLinkedList<T> {
    class Node<T> {
        ... }

    Node<T> front       // points to the front of the list
    Node<T> back        // points to the back of the list
    int     size         // the number of elements in the list

    void add(T item, int index) {
        if((index<0) || (index>size)) throw IndexOutOfBoundsException;
        if(size == 0) front = back = new Node(item, null, null);
        else if(index == 0) front = new Node(item, front, null);
        front.next.prev = front;
        else if(index == size) back = new Node(item, null, back);
        back.prev.next = back;
        else tmpNode = getNth(index-1);
        tmp.next.prev = new Node(item, tmp.next, tmp);
        tmp.next = tmp.next.prev;
        size++
    }
}
```
Remove will require similar modifications.

A LL with a **dummy node** is a doubly LL such that there is always one unused “dummy node” at the front of the list so that you never have null pointers for the front and back list pointers. Adds and removes complexity.

A **circular LL** is a doubly LL with only one pointer into the list, front, and the last node’s next pointer is front, and front.prev = last node. I.e. forms a circle!

In summary:
- doubly LL’s add a bit more complexity to your code (more pointers)
- but they allow traversals in both directions
- adding to the front and back are now both O(1) operations
- removing from the front and back are now O(1) operations, too
- however add and remove are still O(n) when removing from the middle
- and that isn’t going to go away, unfortunately
- you can get rid of some of the case-work with the pointers by using a dummy node, or a circular linked list, but they add their own cases, too...
An Iterator is a structure that allows you to step through it sequentially.

The Iterator interface contains 3 methods:

- boolean hasNext()  // return true if elements remain to iterate
- T next()            // returns the next element, if none throw exc.
- void remove()       // removes last element returned by next()

You’ve used iterators, for example, a Scanner is an iterator:

Scanner input = new Scanner( new File( “test1.txt” ) );
while( input.hasNext() )
    System.out.println( input.next() )

Each Iterator you create is keeping its own pointer into the relevant data sequence (in the case above, a file). If you make two Scanners, they have their own pointers and can iterate through the file at different rates.

When you implement a data structure that stores sequential data, it is nice to provide an Iterator so that users can easily iterate through the data using this same code pattern (and they don’t have to learn the data structure’s terms).
// For ex, you may create an ArrayListIterator inside ArrayList:

class MyArrayListIterator<T> implements Iterator<T> {
    private int index;   // what index will you iterate next

    public MyArrayListIterator() {
        index = 0;        // Initially you’re at the start
    }

    public boolean hasNext() {
        return (index < size);  // You’re not yet at the end
    }

    public T next() {
        if( hasNext() ) {
            return get(index++); // increments index *after* get
        }                       // this calls ArrayList’s get(..)
        throw new NoSuchElementException();
    }

    public void remove() {
        throw new UnsupportedOperationException(); // not yet :-)
    }
}
And now I want to use the ArrayListIterator to iterate over my array list...

```
MyArrayList<String> mal = new MyArrayList<String>();
mal.add( "Hello." );
mal.add( "fuzzy" );

// But how do I use my iterator? How do I even get my iterator?
```

We should really have a method in the MyArrayList class that returns a MyArrayListIterator, so that I can get one easily!

```
MyArrayListIterator<String> mali = mal.iterator();
while( mali.hasNext() )
    System.out.println( mali.next() )
```

// Good... but there are still a few problems...
// For one, can I see MyArrayListIterator from the outside?
// And if these elements are iterable, shouldn’t this kind of method be standard?
Once you have an Iterator, it’s useful to provide access to it!

The `Iterable` interface has a “factory method” `iterator()`

```java
public class MyArrayList<T> extends AbstractList<T> implements Iterable<T> {
  T[] data;
  int size;    // same as before
  ... <class methods as before>

  public Iterator<T> iterator() {
    return new MyArrayListIterator<T>(this);
  }
}
```

You could modify your `ArrayList` class to implement `Iterable<T>` so that you can produce iterators to iterate over the elements of your arraylist.

```java
public class MyArrayList<T> extends AbstractList<T> implements Iterable<T> {
  T[] data;
  int size;    // same as before
  ... <class methods as before>

  public Iterator<T> iterator() {
    return new MyArrayListIterator<T>(this);
  }
}
```
In summary, if your class has iterable elements, you should:

1. create an inner class that implements an Iterator<T>, to iterate over elements
2. have the data structure implement the Iterable<T> interface, which needs...
3. implement the iterator() method in the DS to return instance of your Iterator

If you do this, then code like this will work:

```java
MyArrayList<String> mal = new MyArrayList<String>();
/* Maybe add some stuff to mal so it's not totally boring... */

Iterator<String> it = mal.iterator(); // get the iterator
while( it.hasNext() )
    System.out.println( it.next() ) // print out all elements

/* Even better, you can use the fancy for loops */
for( String s : mal )
    System.out.println( s )
```