Announcements

- 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.

Linked List Review

Recall that a linked list is a data structure that represents a sequence of elements that are stored non-contiguously in memory.

- 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).

Doubly Linked List Review

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.

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
        }
        public Node<T> getPrev() return prev
        public void setPrev( Node<T> 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 ) {
        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.

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 O(1) operations, too
- removing from the front and back are now O(1) operations
- doubling the list and removing the last element are O(n) operations
- adding to the front and back are now O(1) operations
- but they may involve invalidation of other elements
- A circular LL is a doubly LL with only one pointer into the list, front, and the back
- the front and back pointers, adds and removes complexity
- A LL with a dummy node is a doubly LL such that there is always one unused

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

Iterator Pattern

An Iterator is a structure that allows you to step through it sequentially:
- 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:
```java
Scanner input = new Scanner( new File( "test1.txt" ) );
while( input.hasNext() )
    System.out.println( input.next() )
```

When you implement a data structure that stores sequential data, it is nice to provide an Iterator so that users can easily traverse through the data using this same code pattern (and they don't have to learn the data structure's terms)

Doubly Linked List Implementation

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

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

For example, you may create an ArrayListIterator inside ArrayList:

```java
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
        } else {              // this calls ArrayList's get()
            throw new NoSuchElementException();
        }
    }
    public void remove() {
        throw new UnsupportedOperationException(); // not yet :-)
    }
}
```

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

// Good... but there are still a few problems...