Announcements

Prelab 4 due on Monday by 10am

Lab 3: I hear some of you don’t remember try-catch statements! Go back to lecture 3 to read up on it.
ArrayList Running Times

Recall that all ArrayList operations are $O(1)$ except for add and remove, $O(n)$.

- `int size()` -- $O(1)$
- `boolean isEmpty()` -- $O(1)$
- `void clear()` -- $O(1)$
- `T get(int index)` -- $O(1)$
- `T set(int index, T element)` -- $O(1)$
- `T remove(int index)` -- $O(n)$ (worst case: index=0 & need shift)
- `add(int index, T element)` -- $O(n)$ (worst case: same as remove)
- `add(T element)` -- always adding to end... so $O(1)$...

  unless there is a resize requiring $O(n)$ work.

  but by double resizing, over n adds we only do $O(1)+O(2)+O(4)+O(16)+...+O(n)=O(n)$ work.

  Thus, add is amortized (i.e. average) $O(1)$
Recall that all Stack and Queue operations are $O(1)$

- int size() -- $O(1)$
- boolean isEmpty() -- $O(1)$
- void clear() -- $O(1)$
- T pop() / T top() / void push(T element) -- $O(1)$
- T dequeue() / T front() / void enqueue(T element) -- $O(1)$

- remember that if we use an array-based implementation then both the push and enqueue methods are amortized $O(1)$, as they require the occasional double resize.

- alternatively, we can used a linked implementation, where each “node” in the list keeps track of the element itself as well as the next node in the list. This allows for dynamic resizing of our lists and (non-amortized) $O(1)$ operations.
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

These are pretty much the same operations as for an ArrayList, but we will implement it differently and consequently get different run times.

We will get rid of the amortized behaviour of the ArrayList, but our worst-case behaviour will not improve, unfortunately.
Our linked list class will have to contain a private class representing Nodes:

class LinkedList<T> {
    class Node<T> {
        private T data
        private Node<T> next

        public Node( T data, Node<T> next )
            this.data = data
            this.next = next
        public T getData() return data
        public Node<T> getNext() return next
        public void setData( T data ) this.data = data
        public void setNext( Node<T> next ) this.next = next
    }
    Node<T> front // points to the front of the list
    int size // the number of elements in the list

    // Linked List class methods go here
}
The size(), isEmpty(), and makeEmpty() methods are simple, as usual:

class LinkedList<T> {
    class Node<T> {...}

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

    int     size()          return size           // O(1)
    boolean isEmpty()       return (size() == 0)  // O(1)
    void    makeEmpty()     size=0; front = null  // O(1)
}
For add( item, index ) we may have special cases to be careful of:

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

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

    void add( T item, int index ) {
        if( (index < 0) || (index >= size) ) throw exception
        if( index == 0 ) // special case for add to front
            front = new Node( item, front )
        else
            Node<T> tmp = front
            for i=1 to index-1      // this loop gets (index-1)st node
                tmp = tmp.next
            tmp.next = new Node(item, tmp.next)
            size++
    } // Runtime is O(n) worst-case (e.g. adding to the end)
}
```
For `remove(index)` we may also have special cases to be careful of:

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

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

    T remove( int index ) {
        if((index < 0)|| (index >= size)|| isEmpty()) throw exception
        if( index == 0 ) // special case for remove from front
            tmp = front.getData()  // get the data from the node
            front = front.next()     // remove the front node
        else
            Node<T> prev = getNthNode( index - 1 )
            tmp = prev.getNext().getData()  // get Data of index node
            prev.setNext( prev.next().next() ) // remove the node
            size--
            return tmp
    } // Runtime is O(n) worst-case (e.g. removing from the end)
}
```
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:

- **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

But our implementation will be slightly different since we now have two pointers into our linked list instead of just one.
A doubly-linked list needs to have Nodes that have next and previous pointers:

class LinkedList<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
    }

    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

    // Doubly Linked List class methods go here
}

Thursday, September 27, 12
Doubly Linked List Implementation

For add( item, index ) we may have special cases to be careful of:

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

    Node<T> front        // points to the front 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?
    }
```
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()

Where have you seen an iterator before? What class has these methods, and therefore likely implement this interface?

When you create a data structure that stores elements, such as an arraylist or linked list, it is good practice to also provide an iterator class that iterates over those elements (without changing the elements).

Usually you just place it as another non-public class in the same class as your public data structure. This is called an inner class, fyi. This way, this class has access to all of the class members and class methods of the structure, but itself represents a specific iterating structure. (Same as the Node<T> class!)
// 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;
    }

    public boolean hasNext() {
        return (index < size);
    }

    public T next() {
        if( hasNext() ) {
            return get(index++); // increments index *after* get
        }
        throw new NoSuchElementException();
    }

    public void remove() {
        throw new UnsupportedOperationException(); // not yet :-)
    }
}
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(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(this);
    }
}
```

Now, your ArrayList is Iterable! And it can be used in those funky for-loops, automatically...
In summary, if your class has iterable elements, you should

1. implement an Iterator class that implements `Iterator<T>`,
2. implement the `Iterable<T>` interface, which requires you to
3. implement the factory method `iterator()` 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( it.next() ) } 
```