Announcements

Prelab 4 due on Monday by 10am

Lab 2: I hear some of you don't remember try-catch statements! Go back to lecture 3 to read up on it.

ArrayList Running Times

- 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 ArrayList operations are O(1) except for add and remove, O(n).

Stack & Queue Running Times

- 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)

Recall that all Stack and Queue operations are O(1)

- remember that if we use an array-based implementation then both the push and the pop are O(1)
- if we use a linked implementation then both the push and the pop are O(n)
- if we use a linked implementation then both the push and the pop are O(1)
- 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 ArrayList operations are O(1) except for add and remove, O(n).

Linked Lists and Iterators

CS 151
A linked list is a data structure that represents a sequence of elements that are stored non-contiguously in memory. When implementing a linked list, a special case for add to front must be handled:

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

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 = null; // points to the front of the list
    int size = 0;

    public LinkedList() {
        // constructor code
    }

    public void add(T item) {
        if (front == null) {
            front = new Node<>(item, null);
            size++;
            return;
        }
        Node<T> current = front;
        while (current != null) {
            if (current.getNext().getData().equals(item)) {
                current.setData(item);
                return;
            }
            current = current.getNext();
        }
        Node<T> newNode = new Node<>(item, null);
        current.setNext(newNode);
        size++;
    }

    public T remove(T item) {
        if (front == null) {
            return null;
        }
        Node<T> current = front;
        Node<T> previous = null;
        while (current != null) {
            if (current.getData().equals(item)) {
                if (previous != null) {
                    previous.setNext(current.getNext());
                } else {
                    front = current.getNext();
                }
                size--;
                return current.getData();
            }
            previous = current;
            current = current.getNext();
        }
        return null;
    }

    // Additional supported methods
}
```
Linked List Implementation

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)
}
```

Doubly Linked List

A doubly-linked list is a linked list where each node not only keeps track of:

- The next element in the list, but also 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

- void clear() { head = null; } // clear all items from the stack
- int size() { return the number of elements in the stack
- int locate(item) -- returns the index of the specified item in the stack
- void remove( item ) -- remove the item at the specified index
- void add( item ) -- add the item at the specified index

The next element in the list, just like the previous one.

A doubly-linked list

Doubly-Linked List Implementation

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

```java
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
}
```

Doubly-Linked List Implementation

For remove(index) we may also have special cases to be careful of:

```java
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
}
```
**Iterator Pattern**

An Iterator is a structure that allows you to step through it sequentially.

```java
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?

The Iterator interface contains 3 methods:

1. `hasNext()`: Returns `true` if there are more elements in the iterator.
2. `next()`: Returns the next element in the iteration. Throws an exception if there are no more elements.
3. `remove()`: Removes the last element that was returned by the `next()` method. This is optional and may or may not be supported by the implementation.

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. This way, this class has access to all the members and methods of the structure, but itself represents a specific iterating structure. (Same as the Node<T> class!)

```java
public 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!

```
public class MyArrayList<T> extends AbstractList<T>
    implements Iterable<T> {
    T[] data;    // 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. Implement an Iterator class that implements `Iterator<T>`,
2. Implement the `Iterable<T>` interface, which requires you to implement the factory method `iterator()` to return an instance of your Iterator,
3. If you do this, then code like this will work:

```
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 may create an ArrayListIterator instance like this: */
```

**Conclusion**

Now your `Arraylist` is iterable and can be used in those `for` loops.

```java
for (String s : mal) {
    System.out.println(s);
}
```

In a nutshell, if you have a collection of `Element<T>` objects stored in your collection, you can iterate over those elements using `Iterator<T>` objects. You could modify your `Arraylist` class to implement `Iterator<T>` so that you can produce iterations to return the elements of your `Arraylist`.

When you create a collection that stores elements, such as an arraylist, you can produce iterators to return your collection. You could modify your `Arraylist` class to implement `Iterator<T>` so that you can produce iterators to return the elements of your `Arraylist`.

The `Iterator` interface is a structure that allows you to step through elements sequentially.

An Iterator is a structure that allows you to step through elements sequentially.