Implementing
Stacks, Queues and Lists
Over the rest of the semester we will implement a variety of structures. We usually want to be able to say “new X” and get a structure of type X, so that means X needs to be a class. We will usually want to make the process of creating a structure separate from adding data to it, so we can’t just use null to represent an empty structure. For each structure the questions are: how do we create an empty structure? How do we add data to the structure? How do we remove data from the structure?
For dynamic structures we will usually make the Node structure as a class nested within the larger class. For example, if we wanted to make a linked list class we might start

```java
public class MyLinkedList<E> {
    Node head, tail, etc.  // List variables
    protected class Node {
        E data;
        Node next;
        // Node methods come here
    }
    // List methods  come here
}
```

// List methods come here
Stacks
Remember that stacks are “Last in, first out” structures.

For an ArrayList implementation we could do this:

To construct a Stack we will make an ArrayList \textit{data}. The stack’s \texttt{push( )} operation corresponds to the ArrayList’s \texttt{data.add( )}. The Stack’s \texttt{pop} operation corresponds to \texttt{data.remove(size()-1)}. The Stack’s \texttt{size()} operation is just the ArrayList’s \texttt{size()} operation; the Stack is empty when the ArrayList is empty.
For example, if we create a new Stack and push 23, 34, and 45 on it in that order, it will look like this:

<table>
<thead>
<tr>
<th>data</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>&quot;&quot;</th>
</tr>
</thead>
<tbody>
<tr>
<td>23</td>
<td>34</td>
<td>45</td>
<td>....</td>
<td></td>
</tr>
</tbody>
</table>

Size == 3
Code for this looks like this:

```java
public class MyStack<E> {
    private ArrayList<E> data;

    public MyStack() {
        data = new ArrayList<E>();
    }

    public push(E item) {
        data.add(item);
    }

    ....
}
```
Alternatively, to create a dynamic structure for a Stack we will use a Node class with attributes *data* and *next*. The Stack class has a variable *Node top*; that represents the top of the stack. For an empty stack, *top* is just *null*.

Here is a picture of a dynamic stack where we started with the empty sack, and then pushed values 23, 34, and then 45:

```
   45  top
    ↓
   34
    ↓
   23
```

Note the direction of the arrows; we need to be able to follow the links from top to top.next to top.next.next, and so forth.
The code starts like this:

```java
public class MyStack <E> {
    Node top;
    int size;
    protected class Node {
        E data;
        Node next;
        ....
    } // end of Node class declaration

    public MyStack() {
        size = 0;
        top = null;
    }
    ...
}
```
Queues are “First in, first out” structures. They enter at one end of a structure and leave from the other.

For an ArrayList implementation we might have data enter at the far end of the list (so \texttt{enqueue(x)} corresponds to \texttt{data.add(x)}) and exit from index \[0\] (\texttt{data.get(0)}).
Here is a picture of the ArrayList we would get from starting with an empty queue and enqueuing 23, then 34, then 45

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>23</td>
<td>34</td>
<td>45</td>
</tr>
</tbody>
</table>

Size == 3

The difference between the structures lies in how we do the remove operation. Popping the stack results in

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>23</td>
<td>34</td>
</tr>
</tbody>
</table>

While dequeuing the queue gives

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>data</td>
<td>34</td>
<td>45</td>
</tr>
</tbody>
</table>
For a linked implementation of a Queue we might use this:

```
[23 -> 34 -> 45]
```

A dequeue operation corresponds to `front=front.next`, while `enqueue(x)` is
```
Node p = new Node(x);
back.next = p;
back = p;
```

The empty queue will be represented by both `front` and `back` being null.
Note that this Queue representation requires us to make a special case out of the empty queue: if we start with front and back both being null, the first time we enqueue something we have to set both front and back to that new node. If we dequeue a Queue with just one entry we need to make both front and back be null.

We can avoid these special cases if add one extra node to structure as a placeholder. I will call this node “head” instead of “front”.
We start an empty Queue with just the head node:

```
head   back
```

We always dequeue with `head.next = head.next.next`. We always enqueue(x) with

```
Node p = new Node(x);
back.next = p;
back= p;
```

The only special case is that when we dequeue the last node in the queue we need to set `back=head`
Note that we could use this last structure for a general-purpose linked list. I usually call the ends of the list “head” and “tail” rather than “head” and “back”. We don’t put data in the head node; it is just there as a placeholder. Some people refer to such placeholder nodes as “sentinel” nodes. We can find the ith node by starting at the head and following i+1 next links. For example, the node at index 2 is head.next.next.next.

We will create a structure like this in Lab 4.