administrivia...
-assignment 6 due tonight at midnight

-assignment 7 is out
midterm scores

<table>
<thead>
<tr>
<th>Score Range</th>
<th>Number of Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>0-10</td>
<td>0</td>
</tr>
<tr>
<td>11-20</td>
<td>0</td>
</tr>
<tr>
<td>21-30</td>
<td>0</td>
</tr>
<tr>
<td>31-40</td>
<td>1</td>
</tr>
<tr>
<td>41-50</td>
<td>5</td>
</tr>
<tr>
<td>51-60</td>
<td>19</td>
</tr>
<tr>
<td>61-70</td>
<td>35</td>
</tr>
<tr>
<td>71-80</td>
<td>54</td>
</tr>
<tr>
<td>81-90</td>
<td>45</td>
</tr>
<tr>
<td>91-100</td>
<td>15</td>
</tr>
</tbody>
</table>
last time...
-a stack is a data structure in which insertion and removal is restricted to the top (or end) of the list

-also called FIRST-IN, LAST-OUT (FILO)
  -insertion always adds an item to the end
  -deletion always removes an item from the end
-push, pop, and peek must all be O(1)

we need a very efficient data structure if we expect to only access the last element

HOW CAN WE IMPLEMENT A STACK SO THAT ALL 3 OPERATIONS ARE GUARANTEED TO BE O(1)?
as an array...

push (a)  push (b)  pop ()

top=−1  top=0  top=1  top=0
as a linked list...

null

push (a)

push (b)

pop ()
EXAMPLE: symbol matcher
for (i=0; i<N; i++)
{
    arr[i] = i;
}

today...
- ANOTHER STACK EXAMPLE: postfix notation
- queues
- priority queues
- homework 7 hints
EXAMPLE: postfix notation
-we usually see expression written in **infix notation**

- place an *operator* in between a left and right *operand*
  - \( a + b \)

-the order of operations is not clear from the expression without parentheses
  - although, left-to-right is often assumed
  - \( -1 + 2 * 3 = ? \)
  - *answer is 7, but some calculators will give 9!*
postfix expressions

-a syntax lacking parentheses that can be parsed without ambiguity
-also called reverse polish notation

to operands, followed by an operator

```
 a b +
```

```
1 2 3 * +
```

→ $2 \times 3$ is evaluated first, result is then added to 1
HOW CAN WE USE A STACK TO EVALUATE A POSTFIX EXPRESSION?

1 2 3 * + 4 -
(ANSWER IS 3)

HINT:
- when an operand is seen, __________
- when an operator is seen, __________
- when the expression is done, __________
-when an operand is seen, **push it onto the stack**

-when an operator is seen, **the right and left operands are popped, the operation is evaluated, and the result is pushed back onto the stack**

-when the expression is done, **the single item remaining on the stack is the answer**
1 2 3 * + 4 -

operator operator operator
push(push(pop()), pop(), push(r))

2 * 3 = 6
1 2 3 * + 4 -

operator and operand
pop(), pop(), push(r), push(r)

6
7

1 + 6 = 7
\[ 1 \ 2 \ 3 \ \ast \ + \ 4 \ - \]

\text{operator\ EOL}

\text{pop()}, \ \text{pop()}, \ \text{push}() \ \text{pop}() \\
\text{ANSWER IS} \ 11
queue
- A **queue** is a FIRST-IN, FIRST-OUT data structure - FIFO

- Insert on the back, remove from the front

- Operations:
  - **enqueue**... adds an item to the back of the queue
  - **dequeue**... removes and returns the item at the front

  **Terminology avoids confusion with a stack!**

- Like a stack, all operations are $O(1)$
enqueue(8)

front \[11\ 5\ 2\ 14\ 8\] back
enqueue (8)
dequeue ()
enqueue(8)
dequeue()
enqueue(7)

front 5 2 14 8 7 back

How can we implement a queue so that all operations are guaranteed to be O(1)?
as an array...

- keep track of front and back indices

- front and back advance through the array
  - enqueueing advances back
  - dequeueing advance front

- what happens when back reaches the end of the array?
enqueue(3)

enqueue(6)
performance

- using wrap-around, all operations are $O(1)$ on average

- but, $O(N)$ array growing is still a problem in the worst case!

- how do we hand array growth if there is wrap-around in the queue?
  - how do we hand copying?
  - this is non-trivial…
as a linked list...

-remember, inserting and deleting to the head and tail of a linked list is automatically O(1)

- **front** is analogous to **head**
- **back** is analogous to **tail**

-no messy wrap-around, or growth issues

-which linked list operations are analogous to **enqueue** and **dequeue**?
summary

-linked lists and wrap-around arrays are both $O(1)$ for queue implementations

-BUT, arrays are much more complicated to code

-both queues and stacks require very little code on top of a good linked list implementation
priority queues
-like a queue, but items returned in order of **priority**
  - *dequeue* operation always returns the item with the highest priority
  - if two items have the same priority, the first one in the queue is returned

-how can we implement this?
-can operations be **O(1)**?
using a linked list...

- always add items in correct, sorted spot

enqueue (10)

- dequeue will return smallest item $O(1)$

- what is the cost of enqueue?

- we will study a more advanced priority queue later…
homework hints...
-suppose we want to print the `String`

```java
println("this is a quote: "hello");
```

-will this work?
String literals

certain characters in Strings are special cases
```
`
\ (escape character)
to include a quote character, we must escape it
println("this is a quote: \\
\"");

we can also escape the escape character
println("this is a backslash: \\
");
char literals

- checking for a backslash:
  ```
  if(c == '\\')
  ```

- checking for a double quote:
  ```
  if(c == '"')
  ```

- checking for a single quote:
  ```
  if(c == '\''
  ```
public void test()
{
    /*  )  */
    System.out.println(" "} ");
}

// {}()

IS THIS BALANCED?
next time...
-reading
  - chapters 8 and 19 in book
  - chapter 6
    - http://opendatastructures.org/ods-java/

-homework
  - assignment 6 due tonight