administrivia...
- assignment 6 on linked lists due tonight

- assignment 7 is out
  - due the Wednesday after fall break

- midterm grades out by next Monday

- fall break is next week
last time...
doubly-linked lists
nodes have a link to next and previous node
allows for traversal in either forward or reverse order
maintains a tail node as well as a head node
doubly-linked list insertion:
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';
```
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';
newNode.prev = current;
```

![Diagram of a doubly-linked list insertion](image)
doubly-linked list insertion:

```java
newNode = new Node<Char>(());
newNode.data = 'n';
newNode.prev = current;
newNode.next = current.next;
```
doubly-linked list insertion:

```java
doubly-linked list insertion:

newNode = new Node<Character>();
newNode.data = 'n';

newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
```

Diagram:

```
head  a  c  k  o  y  tail
    ^  ^  ^  ^  ^  ^
    |  |  |  |  |  |
    v  v  v  v  v  
    n
```
doubly-linked list insertion:

```java
newNode = new Node<Character>();
newNode.data = 'n';

newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
newNode.next.prev = newNode;
```
doubly-linked list insertion:

newNode = new Node<Character>();
newNode.data = 'n';

newNode.prev = current;
newNode.next = current.next;
newNode.prev.next = newNode;
newNode.next.prev = newNode;

what is the cost of insertion?
A) c
B) \log N
C) N
D) N \log N
E) N^2
F) N^3
doubly-linked list deletion:

```java
current.prev.next = current.next;
current.next.prev = current.prev;
```

![Doubly-linked list diagram](image-url)
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;

head

a

c

k

o

n

y

current
tail
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;

head  a  c  current  k  o  tail  y
doubly-linked list deletion:

current.prev.next = current.next;
current.next.prev = current.prev;
doubly-linked list deletion:

```java
current.prev.next = current.next;
current.next.prev = current.prev;
```

What is the cost of deletion?

- A) c
- B) log N
- C) N
- D) N log N
- E) N^2
- F) N^3
**LinkedList  vs  ArrayList**

**insertion & deletion:**  
(assuming position is known)  
**O(c)**  
**O(N)**

**accessing a random item:**  
**O(N)**  
**O(c)**

- choose the structure based on the expected use
- what is the common case?
- what if insertion / deletion is always from the front / end?
today...
-stacks

-queues

-priority queues

-homework 7 hints
stacks
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
important methods

- **push**
  - inserts an item on to the top of the stack

- **pop**
  - removes and returns the item on the top of the stack

- **peek**
  - returns but does not remove the top of the stack

- consecutive calls to pop will return items in the reverse order that they were pushed
it is useful to think of stacks as standing upright (like a stack of dishes)
it is useful to think of stacks as standing upright (like a stack of dishes)

pop();

\[
\begin{array}{|c|c|}
\hline
\text{top} & 23 \\
87 & 87 \\
2 & 2 \\
10 & 10 \\
9 & 9 \\
\hline
\end{array}
\]
it is useful to think of stacks as standing upright (like a stack of dishes)

pop();
push(5);
performance

- push, pop, and peek must all be $O(c)$

- 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 three operations are guaranteed to be $O(c)$?
as an array...

- NOTE: keep track of a top index

- to push, increment top, then add the item at that index

- to pop, return the item at index top, and decrement top

```
push(a)
push(b)
pop()
```

```
<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>b</td>
<td>a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>a</td>
</tr>
</tbody>
</table>
```

```
top=-1
top=0
top=1
top=0
```
performance

-if we try to \texttt{push} when the underlying array is full, the array must be grown

-any \texttt{push} that requires resizing the array takes $O(N)$ time

-all other operations are constant, \(O(c)\)

-since \texttt{pushes} that resize the array are rare, the average case for push is still \(O(c)\)
as a linked list...

- treat the head as the top of the stack

- to push, add to the beginning of the linked list

- to pop, return the top and remove the first item
performance

- linked lists never incur the penalty of resizing
  - adds to a linked list are always $O(c)$

- no wasted extra array space

- all stack operations are $O(c)$

- a stack can be easily implemented on top of an existing linked list with very little extra code!
EXAMPLE: call stack (again!)
-every time a method is invoked a unique frame is created

-when that method returns, execution resumes in the calling frame

-methods return in reverse order in which they were called

-FILO!

-what method is the first in and last out?
example

call stack

main
example

```
findAnagrams
main
```

call stack
example

call stack

sort
findAnagrams
main
example

call stack

compare
sort
findAnagrams
main
example

call stack

sort
findAnagrams
main
example

call stack

swap
sort
findAnagrams
main
example

call stack

sort
findAnagrams
main
example

call stack

main

findAnagrams

sort

compare
example

call stack

sort

findAnagrams

main
example

call stack

findAnagrams

main
example

call stack

main
example

call stack

println
main
example

call stack

main
EXAMPLE: symbol matcher
-part of the compilation process for Java’s compiler (and many others) is symbol matching

-every { must be matched with a corresponding }
  -same for () and []

-how can we use a stack to determine if all brace symbols are matched?

```java
for (i = 0; i < N; i++)
{
    arr[i] = i;
}
```
for(i=0; i<N; i++)
{
    arr[i] = i;
}
for (i=0; i<N; i++)
{
    arr[i] = i;
}
for(i=0; i<N; i++)
{
    arr[i] = i;
}

push

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

push
for (i=0; i<N; i++)
{
    arr[i] = i;
}

push
for (i=0; i<N; i++)
{
    arr[i] = i;
}

push

push

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

push

pop

push | pop
for(i=0; i<N; i++)
{
    arr[i] = i;
}

if end of input is reached and the stack is empty... all the symbols are balanced!
EXAMPLE: postfix notation
-we usually see expressions 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  
  \[ 1 + 2 \times 3 = ? \]  
  -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*

- two operands, followed by an operator
  \[ a \ b \ + \]

\[ 1 \ 2 \ 3 \ * \ + \]

\[ \rightarrow 2 \ * \ 3 \ is \ evaluated \ first, \ result \ is \ then \ added \ to \ 1 \]
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 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 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 -
push(1)
operand

1 2 3 * + 4 -
operands

push (1)
push(2)
1 2 3 * + 4 -

operand
push(2)

2
1
push(3)

operand
1 2 3 * + 4 -

operator
pop(), pop(), push(r)
1 2 3 * + 4 -

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

2 * 3 = 6
operator
\text{pop()}, \text{pop()}, \text{push(r)}

2 * 3 = 6

\begin{array}{c}
\hline
6 \\
1 \\
\hline
\end{array}
1 2 3 * + 4 -

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

6
1
1 2 3 * + 4 -

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

1 + 6 = 7
1 2 3 * + 4 -

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

7
1 2 3 * + 4 -

operand
push(4)
1 2 3 * + 4 -

operator

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

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

1 2 3 * + 4 −

7 − 4 = 3
1 2 3 * + 4 -

operator

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

3
$1 \times 2 + 3 - 4 \Rightarrow pop()$

answer is 3
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 $O(1)$
Queues often act as buffers to process items in the order they arrive.
enqueue(8)

front 11 5 2 14 8 back
enqueue(8)
dequeue()
enqueue(8)
dequeue()
enqueue(7)
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 $O(\text{c})$?
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)

[14 8 7 10 3]

front   back

enqueue(6)

[6 14 8 7 10 3]

front

back
performance

- using wrap-around, all operations are $O(c)$ 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?
  - this is non-trivial…
as a linked list...

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

-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 $O(c)$ 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 inserted into the queue is returned

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

- always add items in correct, sorted spot

enqueue(10)

- dequeue will return smallest item \( O(c) \)

- what is the cost of enqueue?

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

this is a quote: "hello"

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

-will this work?
String literals

certain characters in Strings are special cases
```
"\n\" (escape character)
```
to include a quote character, we must escape it
```
println(“this is a quote: \“hello\””);
```
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 18 & 19 in book: trees & binary search trees
  - chapter 6
    - http://opendatastructures.org/ods-java/

-homework
  - assignment 6 due tonight
  - assignment 7 due after fall break