CSE 12 – Basic Data Structures
Prof. Christine Alvarado
[Slides borrowed/adapted from slides by Cynthia Lee, Rakesh Varna, & Roshni Chandrashekhar]
Announcements

1. HW4 posted: START NOW!
2. QuizStar down. You’ll have more time for today’s quiz.
3. Eclipse and submission—be careful!
MRUList vs LinkedList on p&p.txt

- What is the most likely explanation for the behavior observed??
What is the time cost of removing an element at the head or at the tail of an N-element List…

If List is implemented using a singly linked list?
Head: _____ Tail (with direct pointer) _____

A. O(1), O(1)
B. O(1), O(n)
C. O(n), O(n)
D. O(n^2), O(n^2)
E. Other/none/more
Mapping between a LinkedList and Stack (Stack “has a” LinkedList)

- What is the time cost of adding or removing an element at the head or at the tail of an N-element List...
  - If List is implemented using a doubly linked list?
    - Head: _____ Tail (with direct pointer): _____
  
  A. O(1), O(1)
  B. O(1), O(n)
  C. O(n), O(n)
  D. O(n^2), O(n^2)
  E. Other/none/more
Map Stack Attributes to ArrayList Attributes and/or Methods

List.head

List.tail = list.size() - 1

Stack.top

<table>
<thead>
<tr>
<th>Stack Attribute</th>
<th>ArrayList Equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>size() – 1</td>
</tr>
<tr>
<td>size</td>
<td>size()</td>
</tr>
</tbody>
</table>

Don’t underestimate the importance of doing this mapping first. Planning now saves time later.
A consequence of that attribute mapping is that a push operation results in adding to the tail of the List:

\[
\text{size()} - 1 + 1 = \text{size()}
\]

- tail of list
- next position beyond tail
- location to “push” the new stack element

<table>
<thead>
<tr>
<th>Stack operation</th>
<th>List operation equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>push( element )</td>
<td>add( size(), element )</td>
</tr>
<tr>
<td>E pop()</td>
<td>E remove( size() - 1 )</td>
</tr>
<tr>
<td>E peek()</td>
<td>E get( size() - 1 )</td>
</tr>
<tr>
<td>int size()</td>
<td>int size()</td>
</tr>
<tr>
<td>boolean isEmpty()</td>
<td>boolean isEmpty()</td>
</tr>
</tbody>
</table>
Stack Implementations, done correctly...

<table>
<thead>
<tr>
<th>Operation</th>
<th>LinkedList (head = TOS)</th>
<th>ArrayList (end = TOS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Push(E element)</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Pop()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Peek()</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Queues
Queues

- **Operations:**
  - enqueue – append an element to the queue.
  - dequeue – remove element from the queue.
  - peek – return element that would be dequeued with the next call to dequeue.

- Enqueue and Dequeue operate at opposite ends of the Queue.
Queues warm up

- It is a good idea to implement a queue with a class that extends LinkedList \( (\text{is-\ a}) \)
  
  A. Yeah, that’s fine
  B. No, that’s not a great idea

Adaptation pattern!
public E dequeue(){
    // potential issue if empty, for now, assume not empty
    E e = array[front];
    <YOUR CODE HERE>
    return e;
}

Select the correct code to insert from below:

A
front++;

B
rear = rear -1;

C
for(int i = 0; i < rear; i++) {
    array[i] = array[i+1];
}
rear = rear -1;

D
None of these are correct

(a) Queue.front is always at 0 – shift elements left on dequeue().
public void enqueue(E element) {
    // potential issue if full, for now, assume room
    <YOUR CODE HERE>
    front++;
}

Select the correct code to insert from below:

A
array[0] = e;

B
array[front] = e;

C
for(int i = 0; i < front; i++) {
    array[i+1] = array[i];
}
array[front] = e;

D None of these are correct
Queues using Arrays

Instead of shifting elements,
I can maintain pointers that move one step.
ArrayQueue: another option

- Neither of those solutions is very good as they both involve *moving all the existing* data elements, which has high time cost.

- Idea: Instead of moving data elements to a fixed position for *front* when removing, let *front* advance through the array.

Hmmmm….what do we do when we now add an element to that queue at the rear? What happens when we remove several elements, and *front* catches up with *rear*?...
**Solution**: Be more creative!

View the array as *circular* and allow both *front* and *rear* to advance through (around) the array.

This will require *no* data movement for enqueues or dequeues!
Design decisions: Where do front and rear point?

Which of these choices will work?

A  
front  rear

B  
front  rear

C  
front  rear

D  Any of these could work
Design decisions: Where do front and rear point?

Which of these choices will work?

A

B

C

D

Any of these could work

It’s your choice, but make sure you know what you’re doing!