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!
MRUList vs LinkedList on p&p.txt
Mapping between single linked list and Stack (Stack “has a” single linked list)

- 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
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

### Stack operation vs. List operation equivalent

<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

A. Yeah, that’s fine
B. No, that’s not a great idea
Queue Design

Which of the following are behaviors inherent in queues, which are decisions we'll make in specifying the behavior of our queue, which are decisions we'll make when implementing our queue.

1. When you dequeue, you get out the oldest element in the queue
2. When you insert, the element is inserted at the beginning of the linked list.
3. When you dequeue an empty queue, it throws an exception.

<table>
<thead>
<tr>
<th></th>
<th>Inherent</th>
<th>Specifying</th>
<th>Implementing</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>B.</td>
<td>2</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>C.</td>
<td>1</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>D.</td>
<td>2</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>E.</td>
<td>None of these is correct</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
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

(b) Queue.rear is always at 0 – shift elements right on enqueue().
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.

Hmm... 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`...
ArrayQueue: Using a *circular* array underlying data structure

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

Select the correct code to insert from below:

A
    front++;  
    if(front == array.length)  
        front = 0;

B
    rear = rear - 1;  
    if(rear < 0)  
        rear = array.length - 1;

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

D
    None of these are correct
public void enqueue(E e) {
    // potential issue if full,
    // for now, assume not full
    //YOUR CODE HERE
    size++;
}

Select the correct code to insert from below:

A
    rear++; 
    if (rear == array.length) 
        rear = 0; 
    array[rear] = e;

B
    rear++ 
    array[rear] = e;

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

D
    None of these are correct
Suppose our array is full and you try to enqueue, your coworker suggests just throwing an exception because it would be easy to implement?

A. Sounds like a good idea.
B. No, you need to resize and copy, fixing front and rear pointers.
C. I have a better idea.
Design decisions: Where do front and rear point?

Which of these choices will work?

A

B

C

D

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

Which of these choices will work?

A

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>12</th>
<th>8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

front  rear

B

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>12</th>
<th>8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

front  rear

C

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>12</th>
<th>8</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

front  rear

D  Any of these could work

It’s your choice, but make sure you know what you’re doing!
Queues using circular Array

Initially empty:

<table>
<thead>
<tr>
<th>Front,rear</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(10)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>rear</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(11)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>11</th>
<th>rear</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Queues using circular Array

Initially empty:

<table>
<thead>
<tr>
<th>Front, rear</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(10)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>rear</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(11)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>11</th>
<th>rear</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

What should be the value of front after the next dequeue?
A. 0  B. 1  C. 2  D. 5
## Queues using circular Array

Initially empty:

<table>
<thead>
<tr>
<th>Front, rear</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(10)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>rear</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>

enqueue(11)

<table>
<thead>
<tr>
<th>10 (front)</th>
<th>11</th>
<th>rear</th>
<th></th>
<th></th>
</tr>
</thead>
</table>

What should be the value of arr[0] after the next dequeue?  
A. 10  B. 0  C. null  D. It doesn’t matter
Queues using circular Array

dequeue()

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 (front)</th>
<th>rear</th>
</tr>
</thead>
</table>

dequeue()

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 or null</th>
<th>Front, rear</th>
</tr>
</thead>
</table>

enqueue(12)

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 or null</th>
<th>12 (Front)</th>
<th>rear</th>
</tr>
</thead>
</table>
Queues using circular Array

<table>
<thead>
<tr>
<th>10 or null</th>
<th>11 or null</th>
<th>12 (Front)</th>
<th>8</th>
<th>3</th>
<th>rear</th>
</tr>
</thead>
</table>

enqueue(20)

What is the value of rear after this enqueue?
A. 5
B. 0
C. 1
D. 2
E. Other
Queues using circular Array

| 10 or null | 11 or null | 12 (Front) | 8 | 3 | rear |

enqueue(20)

How could you detect when rear needs to be wrapped back to 0?
A. if ( rear > arr.size() )
B. if ( rear > arr.size()-1 )
C. if ( rear >= arr.size()-1 )
D. None of these