CSE 373: Data Structures and Algorithms

Lecture 2: Wrap up Queues, Asymptotic Analysis, Proof by Induction

Instructor: Lilian de Greef
Quarter: Summer 2017

Today:

- Announcements
- Wrap up Queues
- Begin Asymptotic Analysis: Big-O
- Proof by Induction

Announcement: Office Hours

- Announced! See course webpage for times
- Most held in 3rd floor breakouts in CSE (whiteboards near stairs)
- Lilian’s additional ”actual office” office hours
  - CSE 220 (a more private environment)
  - During listed times
  - And by appointment! (email me >24 hours ahead of time with several times that work for you)
  - Come talk to me about anything! (feedback, grad school, Ultimate Frisbee, life problems, whatever)

Announcement: Sections

- When & where: listed on course webpage
- What: TA-led…
  - Review sessions of course material
  - Practice problems
  - Question-answering
- Optional, but highly encouraged!

I wouldn't have passed 332 (Data Structures and Parallelism) without regularly going to section! – Vlad (TA)

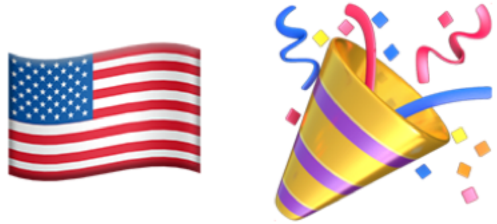
Other Announcements

- Homework 1 is out
  - On material covered in Lecture 1
  - Go forth!
  - …or at least get Eclipse set up today.
- Only required course reading:
  - 10 pages, easy read on commenting style
  - Due beginning of class on Monday
- July 3rd
  - Not an official UW holiday *(sorry guys)*
  - But I'm declaring it an unofficial holiday! Go enjoy a 4-day July 4th weekend

University Holidays

Classes are not in session on the following holidays:

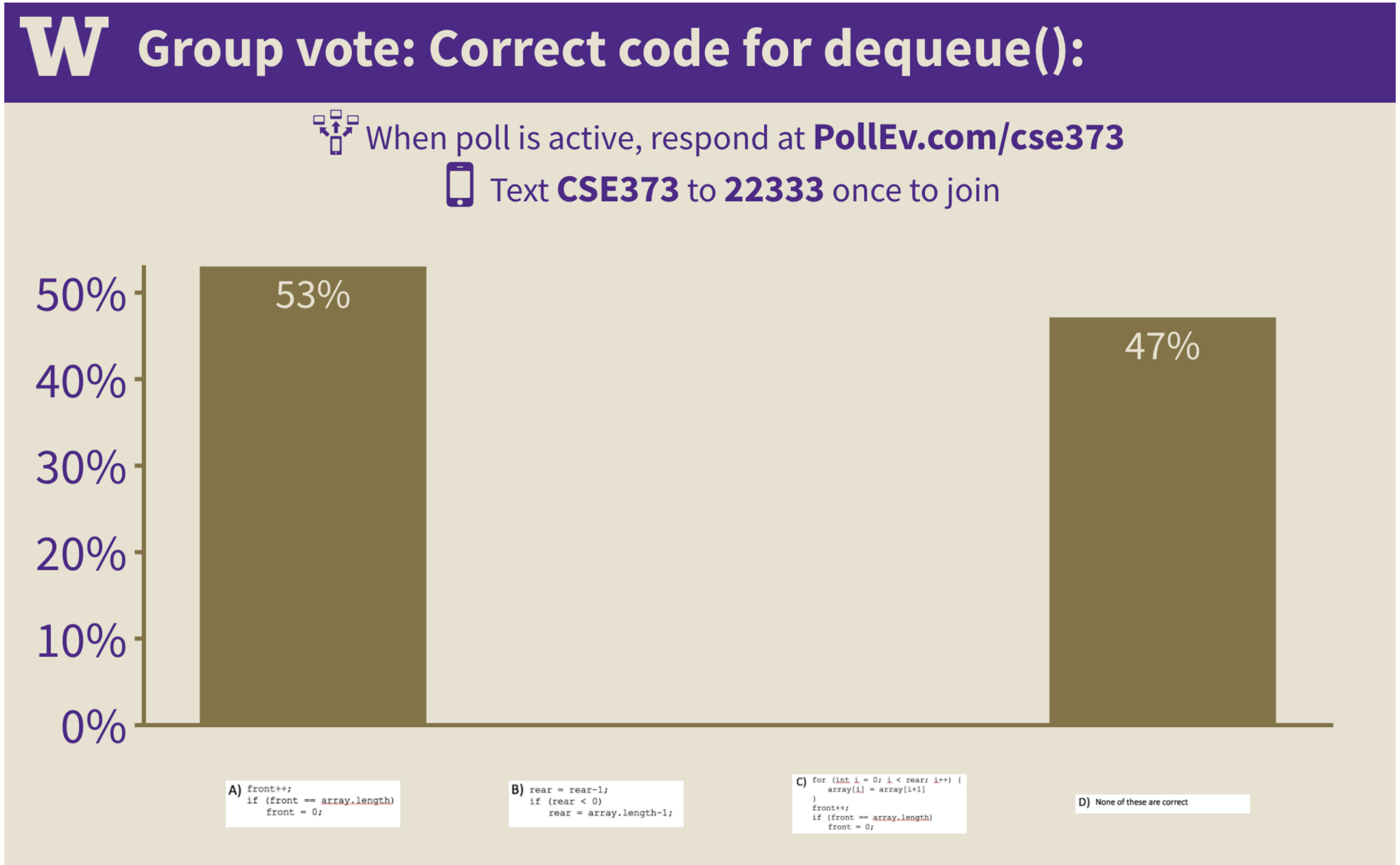
SUMMER 2017		
Full-term	**A-term**	**B-term**
July 4, 2017 Independence Day	July 4, 2017 Independence Day	



Finishing up Queues

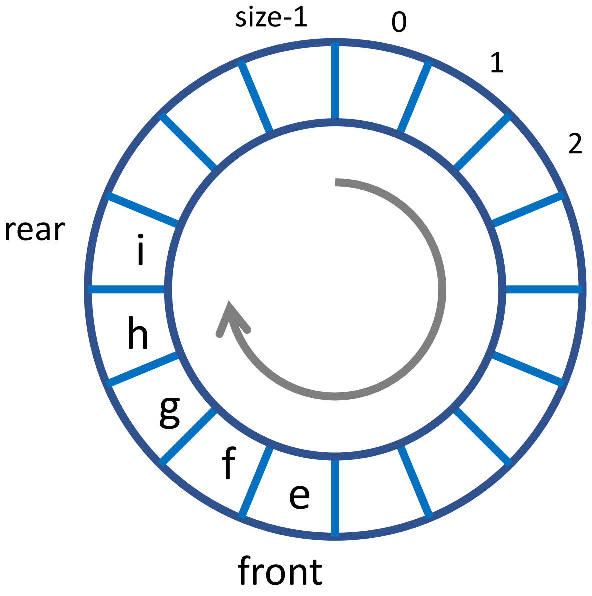
Let’s resolve that cliff-hanger!

Last time, we left off at a cliff hanger…



If we can assume the queue is not empty, how can we implement dequeue()?

```
Public E dequeue() {
    size--;
    E e = array[front];
    <Your code here!>
    return e;
}
```



Handles case where front is at the end of the array

e.g. if front = size++
after "front++"
front = size
not a legal index into the array!

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

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

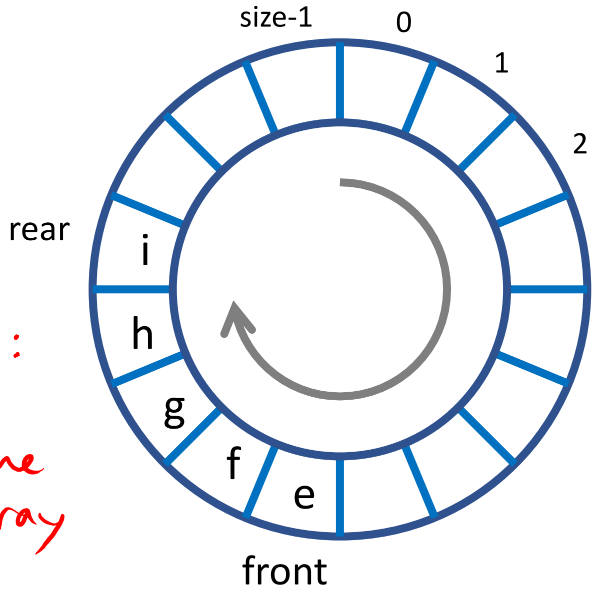
C)
```
for (int i = 0; i < rear; i++) {
    array[i] = array[i+1]
}
front++;
if (front == array.length)
    front = 0;
```

D) None of these are correct

If we can assume the array is not full, how can we implement enqueue(E e)?

```
Public enqueue(E e) {
    <Your code here!>
    size++;
}
```

same as with "front" in dequeue: handles case of rear being at the end of the array



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

B)
```
rear++;
array[rear] = e;
```

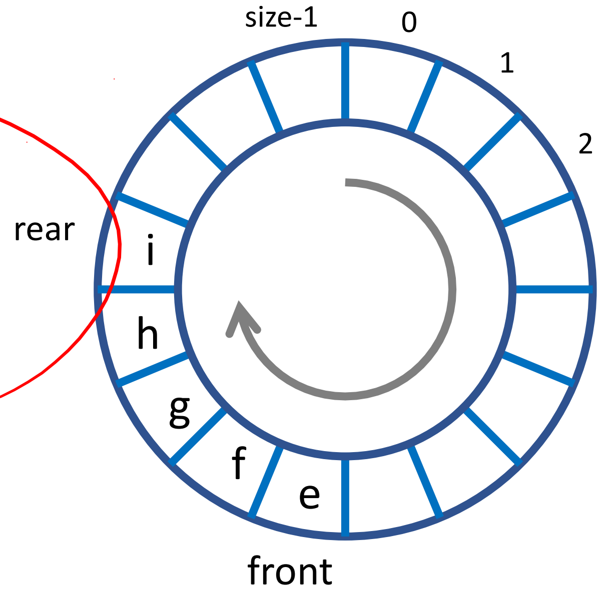
C)
```
for (int i=front; i<rear; i++) {
    array[i] = array[i+1]
}
array[rear] = e;
rear++;
```

D) None of these are correct

If we can assume the array is not full, how can we implement enqueue(E e)?

```
Public enqueue(E e) {
    <Your code here!>
    size++;
}
```

But what if it is full?



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

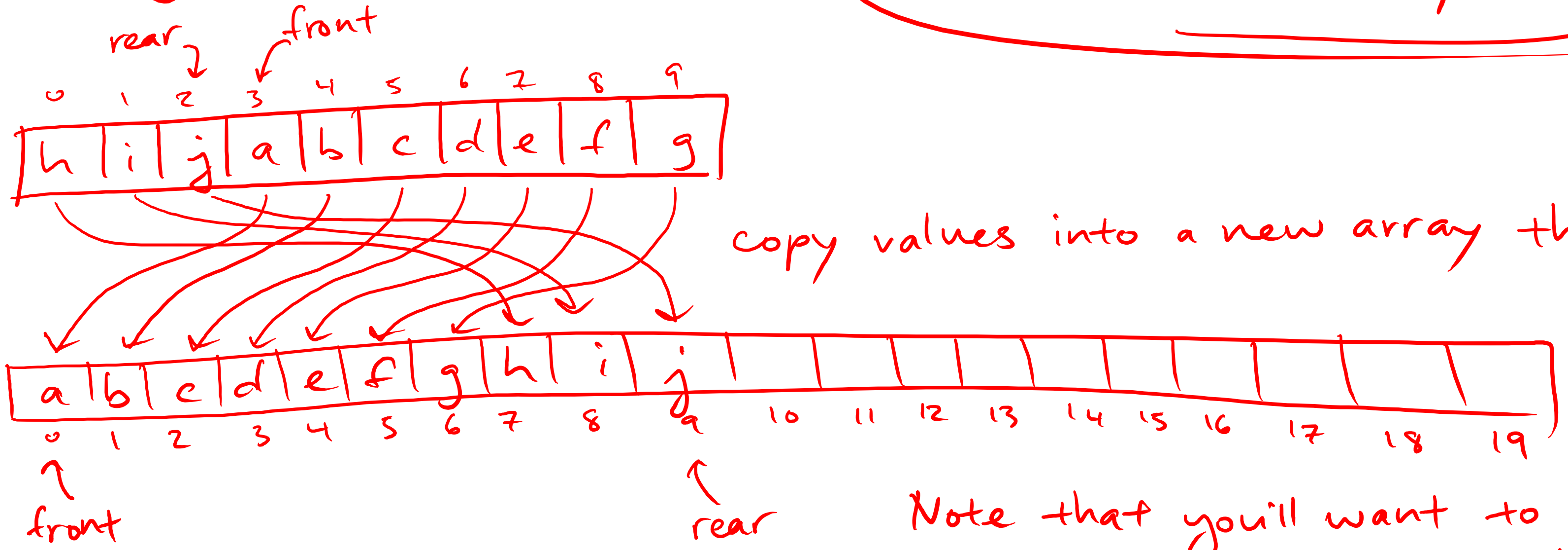
B)
```
rear++;
array[rear] = e;
```

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

D) None of these are correct

Enqueuing to a full array:

Resize the array!

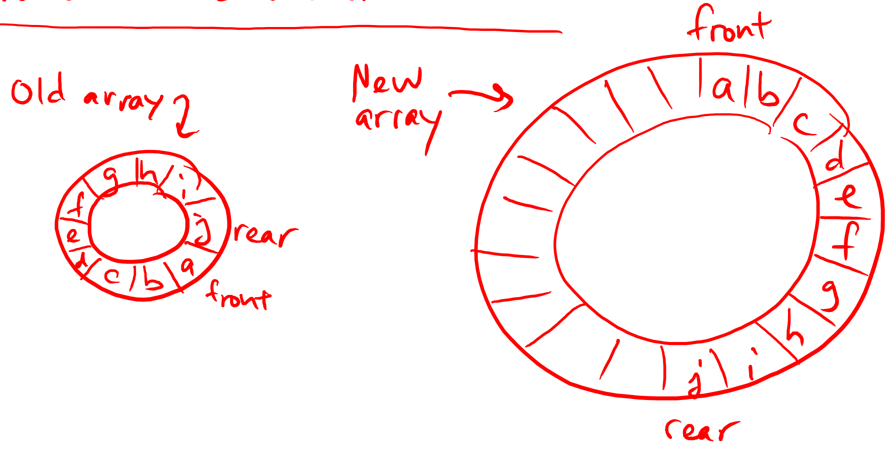


copy values into a new array that's double the size

Note that you'll want to move/copy "front" to be at index = 0 and "rear" accordingly

(why? Try without doing so, and write out what was in the old queue vs the new queue!)

Drawn in circular form:



Between arrays and linked-lists which one *always* is the fastest at `enqueue`, `dequeue`, and `seeKthElement` operations?

(where `seeKthElement` lets you peek at the kth element in the stack)

not the same as average

whether worst-case vs average matters depends on the job!

Fastest:	`enqueue`	`dequeue`	`seeKthElement`
A)	Arrays	Linked-Lists	Neither
B)	Linked-lists	Neither	Neither
C)	Linked-lists	Neither	Arrays
D)	*They're all the same*		

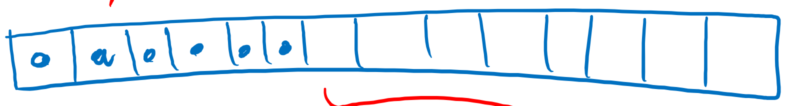
Note: method is not part of Queue ADT. I would not expect queues to have it

Which one's better?

Arrays

- Could get element at arbitrary index k (if needed)
- Uses less memory space per element

Array when not full:



(wasted space)

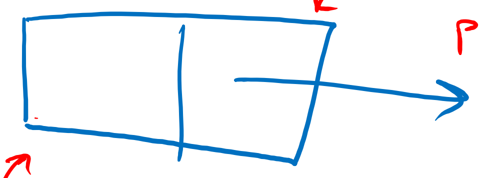
Linked-lists

- Cannot get "full" — no need to resize

linked list nodes have more parts to them than an array cell

Node:

the "next" pointer



container for everything

- Doesn't waste unneeded space, unlike

Trade-offs!

- The ability to choose wisely between trade-offs is why it's important to understand underlying data structures.
- Common Trade-offs
  - Time vs space
  - One operation's efficiency vs another
  - Generality vs simplicity vs performance

Asymptotic Analysis

Oh ho! The Big-O!

Algorithm Analysis

- Why: to help choose the right algorithm or data structure for the job
- Often in **asymptotic** terms

  behavior as a value approaches ∞

- Most common way: **Big-O Notation**
  - General idea: mathematical upper bound describing the behavior of how long a function takes to run in terms of N. ("shape" as $N \to \infty$)
  - A common way to describe "worst-case running time"

Example #1:

The `barn` is an array of `Cows`, excitement is an integer, and `Cow.addHat()` runs in constant time.

```
println("The alien is visiting!");
println("Party time!");
excitement++;
for (int i=0; i<barn.length; i++) {
     Cow cow = barn[i];
     cow.addHat();
}
```

Let's assume that one line of code takes 1 "unit of time" to run
This is not always true, i.e. calls to non-constant-time methods)

Important! Always begin by specifying what “n” is!
(or “x”or “y” or whatever letter)

Example #1:

n = barn.length

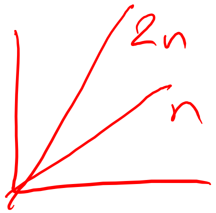
```
println("The alien is visiting!");
println("Party time!");
excitement++;
for (int i=0; i<barn.length; i++) {
     Cow cow = barn[i];
     cow.addHat();
}
```

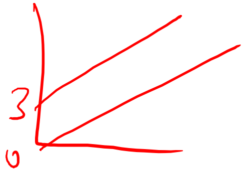
1 +
1 +
1 +
n + (1 + 1)

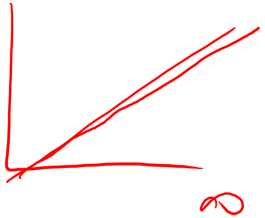
constant

$3 + 2n$ "units of time" $\rightarrow O(n)$

lower order term







(Remember: we care about describing the shape as $n \to \infty$)

Example #2: Your turn!

n = #people in sportsTeam

```
for (Person player: sportsTeam) {
  player.smile();
   for (Person teamMate: sportsTeam) {
     player.say("Good game!");
     player.highFive(teamMate);
}
```

c +

n +

c +

c

n

$$n(c + n(c + c)) = \cancel{cn} + n^2 \cancel{(2c)}$$

$$O(n^2)$$

Assume that the above `Person` method calls run in constant time