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
Finishing up Queues

Let’s resolve that cliff-hanger!
Last time, we left off at a cliff hanger...

Group vote: Correct code for dequeue():

When poll is active, respond at PollEv.com/cse373
Text CSE373 to 22333 once to join
If we can assume the queue is not empty, how can we implement dequeue()?

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

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++;
}

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) {
    size++;
    \textcolor{red}{<Your code here!>}
    rear++;
    \textcolor{red}{array[rear] = e;}
}

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.

Drawn in circular form:

Old array: [a, b, c, d, e, f, g, h, i, j, k, l]

New array: [a, b, c, d, e, f, g, h, i, j, k, l]

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!)
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)

<table>
<thead>
<tr>
<th>Fastest:</th>
<th>enqueue</th>
<th>dequeue</th>
<th>seeKthElement</th>
</tr>
</thead>
<tbody>
<tr>
<td>A)</td>
<td>Arrays</td>
<td>Linked-Lists</td>
<td>Neither</td>
</tr>
<tr>
<td>B)</td>
<td>Linked-lists</td>
<td>Neither</td>
<td>Neither</td>
</tr>
<tr>
<td>C)</td>
<td>Linked-lists</td>
<td>Neither</td>
<td>Arrays</td>
</tr>
<tr>
<td>D)</td>
<td><em>They’re all the same</em></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Linked-lists

- Cannot get “full” — no need to resize
- Linked list nodes have more parts to them than an array cell
- Doesn’t waste unneeded space, unlike

Array when not full:

```
  [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
```

(wasted space)

Node:

```
container for everything
```

the "Next" pointer
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 \( \infty \)

• 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.

```java
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:

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

Constant

\[ 3 + 2n \text{ "units of time"} \rightarrow O(n) \]

Remember: we care about describing the shape as \( n \to \infty \).
Example #2: Your turn!

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

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