CSC 1052 – Algorithms & Data Structures II:

Queues

Professor Henry Carter
Spring 2017

Recap

- Recursion solves problems by solving smaller version of the same problem
- Three components
- Applicable in a range of scenarios
- Not a silver bullet: typically associated with space inefficiency

New ADT: Queues

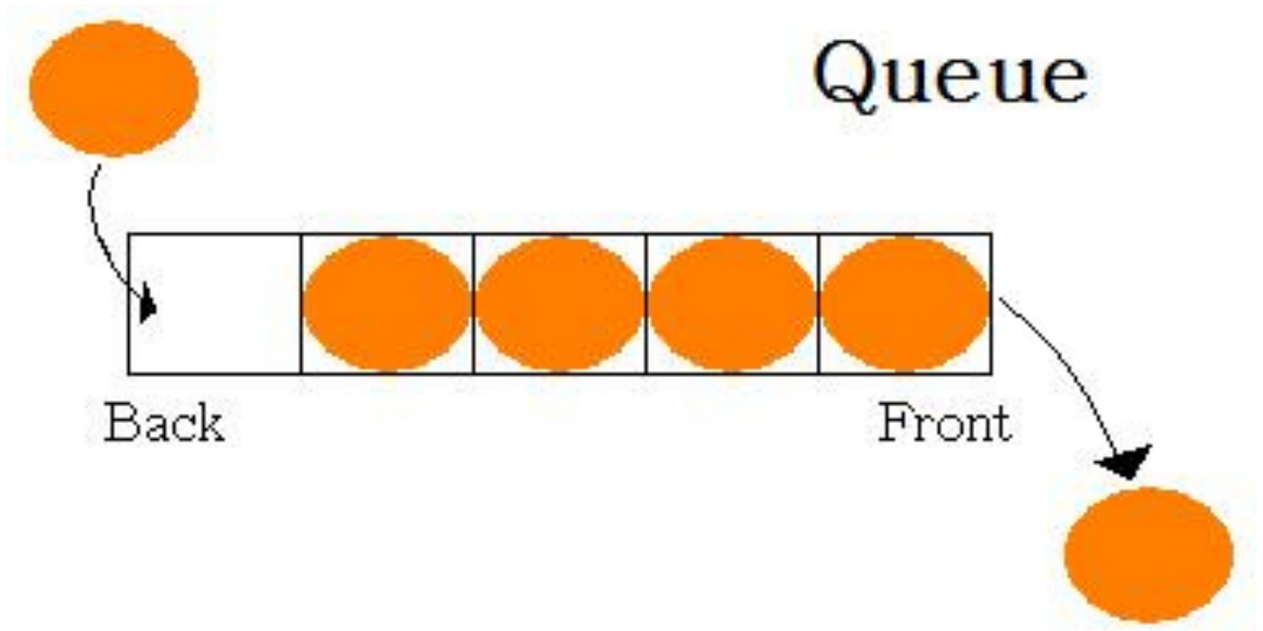
- Waiting in line common in real life and computing
- Data is often processed in the order that it is received
- A queue provides first-in, first-out data ordering (FIFO)
  - Compare to LIFO data structure

Example Queues

- Network packets (buffering)
- Compute jobs/processes
- Support ticketing system

Abstract: The Queue Interface

- Two canonical operations for adding and removing items
  - Enqueue and Dequeue
- Helper operations
  - Size
  - isEmpty
  - isFull



Implementing the Queue

- Create your project
- Import files
- Open the interface

Interface Code

```
package ch04.queues;

public interface QueueInterface<T>
{
   void enqueue(T element) throws QueueOverflowException1;
   // Throws QueueOverflowException if this queue is full;
   // otherwise, adds element to the rear of this queue.

   T dequeue() throws QueueUnderflowException;
   // Throws QueueUnderflowException if this queue is empty;
   // otherwise, removes front element from this queue and returns it.

   boolean isFull();
   // Returns true if this queue is full;
   // otherwise, returns false.

   boolean isEmpty();
   // Returns true if this queue is empty;
   // otherwise, returns false.

   int size();
   // Returns the number of elements in this queue.
}
```

Implementation: Array

- Store items in an array
- Some questions to consider:
  - How do we track the front and back?
  - How large should the queue be?
  - What are the tradeoffs between different techniques?

ArrayBoundedQueue

- Fixed length queue
  - Will require similar overflow/underflow exceptions to our stack
- Front and back indices
- Remaining operations will mirror the stack implementation

Preparation

- Create your ArrayBoundedQueue class file
- Open the interactive test driver
- Try out a few tests (they should all fail!)

Implement!

- Class variables
- Constructors
- Interface methods

Analysis

- Enqueue:

- Dequeue:

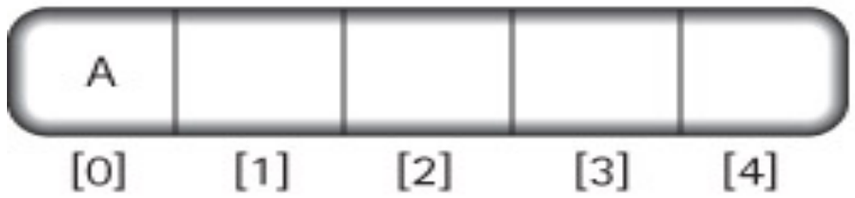
- Helpers:

How do we improve dequeue?

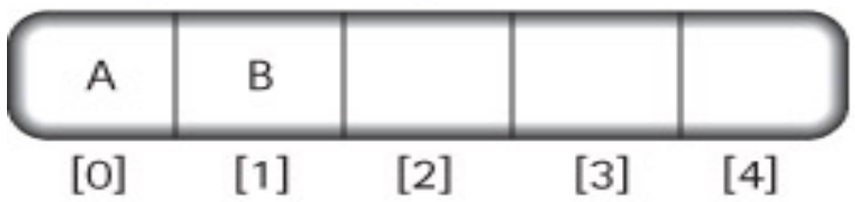
- Functionally, we don't have to start the queue at the beginning of the array
- Track both the beginning and end of the queue elements
- What happens when we hit the end?

Queue wrapping

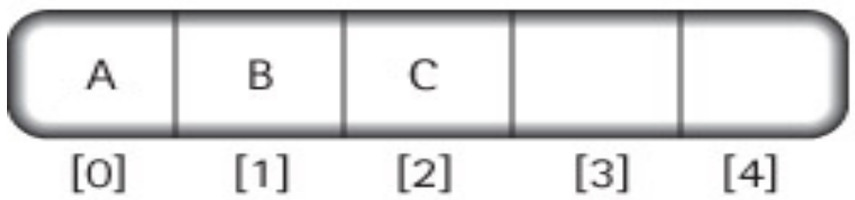
(a) `queue.enqueue('A')`



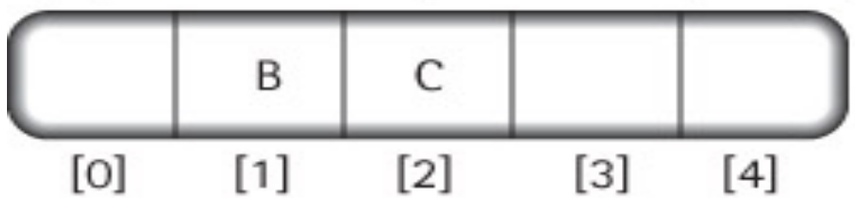
(b) `queue.enqueue('B')`



(c) `queue.enqueue('C')`

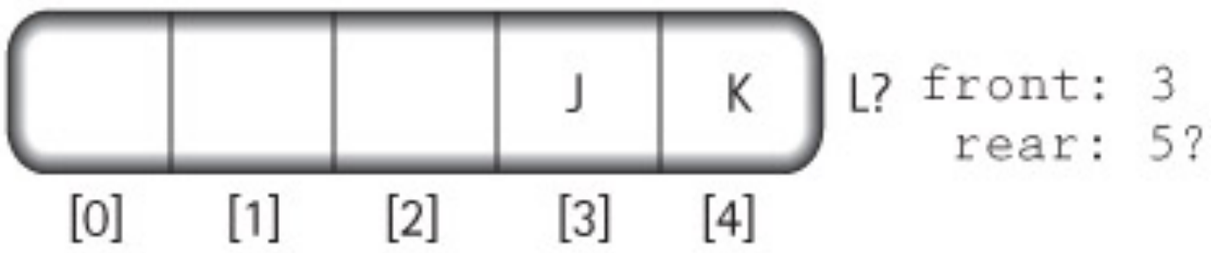


(d) `element=queue.dequeue();`

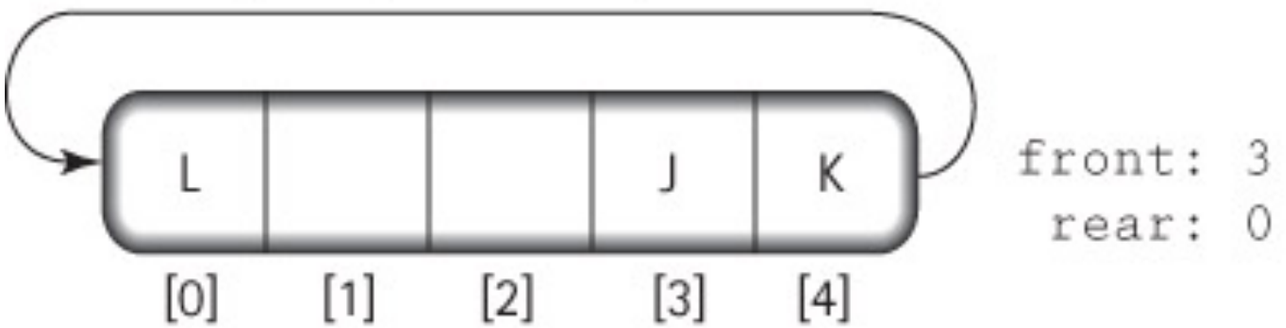


Queue wrapping

(a) There is no room at the end of the array



(b) Using the array as a circular structure, we can wrap the queue around to the beginning of the array



Queue wrapping code

- Modulo operator allows us to wrap the index
- Adds complexity to the code
- Significantly reduces order of growth
  - What is our new OOG?

Implement!

- Modify the ArrayBoundedQueue class to use queue wrapping

Breaking boundaries: unbounded queues

- Functionally the same as a bounded queue
  - Except that the queue is never full
- Primitive arrays are fixed-length in Java
- What do we need to add to get this unbounded functionality?

Enlarge() method

- Creates a new array
  - What size should the array be?
- Copies elements
  - Careful to preserve ordering in case of wrapping
- Replace elements array with new array
  - Swap the pointer!

Enlarge code

```
private void enlarge()
// Increments the capacity of the queue by an amount
// equal to the original capacity.
{
  // create the larger array
  T[] larger = (T[]) new Object[elements.length + origCap];

  // copy the contents from the smaller array into the larger array
  int currSmaller = front;
  for (int currLarger = 0; currLarger < numElements; currLarger++)
  {
    larger[currLarger] = elements[currSmaller];
    currSmaller = (currSmaller + 1) % elements.length;
  }

  // update instance variables
  elements = larger;
  front = 0;
  rear = numElements - 1;
}
```

Cost

- Enlarge worst-case order of growth
- This OOG is passed on to enqueue
- Amortized cost helps us assess the cost of operations that are only performed occasionally
  - If we make many calls to enqueue before enlarging, the cost of one enlarge is amortized
- Evaluated in many different ways
  - Statistically
  - Worst-case
  - Empirically

Recap

- A queue simulates a waiting line, where objects are removed in the same order they are added
- The primary operations are:
  - Enqueue
  - Dequeue
  - Size, isEmpty, isFull
- Array-based implementation requires several technical tradeoffs
  - Fixed front vs floating front
  - Fixed size vs expanding size

Next Time...

- Dale, Joyce, Weems Chapter 4.5
  - Remember, you need to read it BEFORE you come to class!
- Check the course webpage for practice problems
- Peer Tutors
  - http://www.csc.villanova.edu/help/