CS W3134: Data Structures in Java

Lecture #9: Stacks, queues cont'd
10/5/04
Janak J Parekh

Administrivia

- HW#2 went out last week
  - Let's briefly review the homework and what it asks
  - How does one write algorithms without writing code?

Agenda

- Finish stack implementation
- Queue implementation
- Stack/queue examples

Array-based stacks

- Very simple to implement
  - Keep *top* variable, initialized to -1
- Boundary conditions?
- Complexity bounds?
  - Apart from simplicity, biggest reason to use
- Limited size
  - Just like growable lists, we can make growable arrays
  - How?

A stack example

- Delimiter matching: {}()
  - Conceptually simple to use, less error-prone than array
- We'll see a more complex example at the end of class

Queues

- FIFO, instead of LIFO
- "Standing in line": print queue
  - Insert: places at rear of queue
  - Remove: takes from front
  - Peek: looks at front
- Book's convention: front is at bottom, near beginning of array
- Problem: how to represent in array?

Circular queue

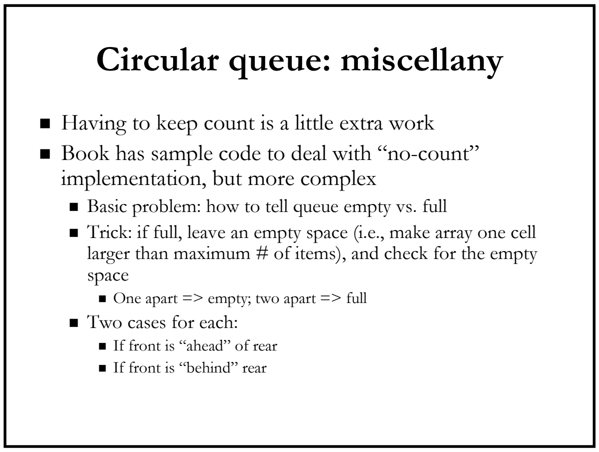
- Don’t move elements around, keep front and back pointers
- Yes, back/front can wrap around: “broken sequence”
- Keep track of number of elements – i.e., full/empty
- Convention: initialize rear to -1, front to 0

Circular queue operations (I)

- Be very careful of keeping pointers consistent
  - Pointers should not “cross” unless empty
- Insert
  - If rear at last element (length-1), reset to -1
  - Increment rear, and then place the object in the new rear
  - Increment # of items
- Remove
  - Grab element at front, and then increment it
  - If front is off the end (== length), reset to 0
  - Decrement # of items

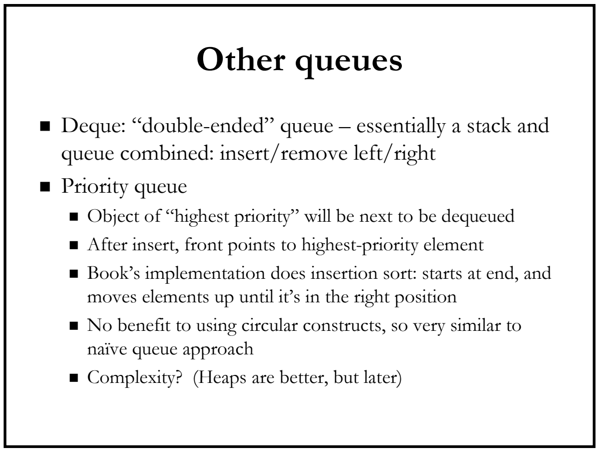
Circular queue operations (II)

- Why -1?
  - Convention so that rear actually points to the newest-added element
  - You can program with 0 if you’re careful
- Efficiency of operations?
- Have you heard of the mod operator?
  - Useful when doing fancy queue manipulations
  - Might want to use it in your homework



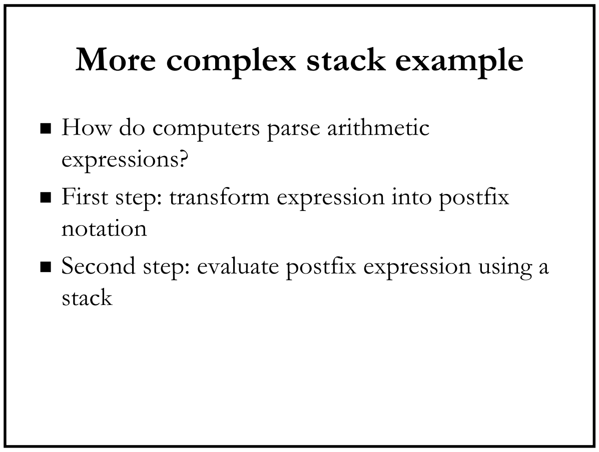
Circular queue: miscellany

- Having to keep count is a little extra work
- Book has sample code to deal with “no-count” implementation, but more complex
  - Basic problem: how to tell queue empty vs. full
  - Trick: if full, leave an empty space (i.e., make array one cell larger than maximum # of items), and check for the empty space
    - One apart => empty; two apart => full
  - Two cases for each:
    - If front is “ahead” of rear
    - If front is “behind” rear



Other queues

- Deque: “double-ended” queue – essentially a stack and queue combined: insert/remove left/right
- Priority queue
  - Object of “highest priority” will be next to be dequeued
  - After insert, front points to highest-priority element
  - Book’s implementation does insertion sort: starts at end, and moves elements up until it’s in the right position
  - No benefit to using circular constructs, so very similar to naïve queue approach
  - Complexity? (Heaps are better, but later)



More complex stack example

- How do computers parse arithmetic expressions?
- First step: transform expression into postfix notation
- Second step: evaluate postfix expression using a stack

Postfix

- Also called Reverse Polish Notation (RPN); HP calculators
- Why?
  - Parentheses unneeded – no ambiguity
  - Can process in one pass from left-to-right
- Fairly straightforward to translate from infix to postfix, but let’s hold off on this

Evaluating a Postfix expression

- Go left-to-right
  - If operand, push on stack
  - If operator, pop two operands, use operator, and push result on stack
- When done, there should be one value on the stack
  - Pop it

Converting Infix to Postfix

- See pages 158-159, although I think these bullets make more sense ;)
- Need to encode *operator precedence*
- To process:
  - Operand: write straight to output
  - (: push on stack
  -): pop all items until (encountered, and output them; don’t write the (
  - Input complete: pop all items and write out
  - Operator: interesting problem

Converting Infix-to-Postfix (II)

- Operator handling
  - If stack is empty, push
  - Else, pop, determine precedence of new vs. popped
    - If popped is a (, put it back on the stack, and put the new operator on top
    - Else if new has higher precedence, push popped back on, and push new on top of it
    - Else if popped has higher or equal precedence, output it, and repeat this process
    - (PE)MDAS for precedence

Next time…

- Linked lists