CS W3134: Data Structures in Java

Lecture #8: Sorts, stacks, queues
9/30/04
Janak J Parekh

Administrivia

- HW#1 due today

Agenda

- Sorting algorithms
- Basic stacks
- Basic queues

(Comparison-based) Sorts

- Bubble (p. 85)
  - Sort pairwise repeatedly
  - Biggest placed each time
- Selection (p. 89)
  - Search for smallest, swap with first
  - Search for smallest, swap with second
- Insertion (p. 95)
  - Take the next one, and put it into the existing sorted subset
- All $O(n^2)$
  - But they're not the exact same performance
- Let's write out a little bit of psuedocode for each

Sorts II

- Lexicographical comparisons?
- Stability of existing items?
- Sidebar: Comparable interface
  - All you have to do is implement boolean compareTo(Object o)
  - Generally a good thing to program to, I prefer to book's example
  - Arrays.sort()

Stacks and Queues

- Useful programmer's tools, will encounter it in many places
  - Very easy and fast to implement
  - Runs very fast as well
- "Restricted access": no index – only manipulate one item at a time
- More abstract – the underlying implementation is unimportant or not remotely similar to the structure, unlike lists

Stacks

- Basic operations: "LIFO" strategy
  - Push
  - Pop
  - Peek
- Analogy: mail basket
  - Not as rigorous as a real stack, of course
- Another analogy: life
  - Conversations
  - Workday
- Extraordinarily simple!

Array-based stacks

- Limited size; ways to get around this
- Decoupled from array index!
- Very simple to implement
  - Keep *top* variable, initialized to -1
- Boundary conditions?
- Complexity bounds?
  - Apart from simplicity, biggest reason to use

Basic Stack examples

- Reverse a word
- Conversation
  - Sentence with parentheses?
- Delimiter matching: {} ()
  - Conceptually simple to use, less error-prone than array
- Function/method calls

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?
  - We can't stick it at one end or the other, unless we slide all the elements around
  - There's a better approach

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?

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

Next time…

- Continue with queues
- More complex examples