CS W3134: Data Structures in Java

Lecture #10: Stacks, queues, linked lists
10/7/04
Janak J Parekh

Administrivia

- HW#2 questions?

Agenda

- Finish queues
- Stack/queue example

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
  - We're not going to do this

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

Linked lists

- Arrays are rather limited, cumbersome data structures – cells are "fixed" together, limited length
- What if we could break apart the cells?
- We *can*!
- In fact, linked list-style structures are used more frequently unless you need very fast random index-based access
- Trees, graphs, etc. are generalizations of linked lists

Linked List structure

- Two basic objects:
  - The list "parent" itself
  - An "element" (book calls "link"), with data
  - Technically, we don't need both
- Parent contains reference to the first element
- *Each element contains a reference to the next element*
- Last element's "next" is set to null

Basic Linked List operations

- How to tell if empty?
- Insertions
  - insertFirst()
  - deleteFirst()
  - displayList()
  - insertLast()
- More complex operations
  - How to find an arbitrary element?
  - How to delete arbitrary element?

Double-ended list

- Contains pointer to last element
- Makes insertLast() much faster (how much?)

Linked list complexity?

- Similar to arrays
- O(1) insert/delete at beginning (or end of list for double-ended)
- Other operations take O(N), but faster than array if "sliding" is needed in array
- Memory?
  - Linked list more efficient, although it has to keep lots of references

Revisit abstraction

- Book finally covers abstraction here
- We can redo all of our previous data structures, previously *array-backed*, as *linked list-backed*
- *Interface* – high-level contract, while the dirty details are hidden
- How to do a stack?
- How to do a queue?
- You should read through this section

Next time…

- Finish Linked Lists
- Start Recursion