Practice of Programming using Java

Lecture 4
June 20, 2006
6-8pm LT
A Problem Solving Architecture

- **Problem Solver**
  - Data Structures
  - Algorithms

- **Main**: glue between problem solver classes, input and output
  - This makes your problem solver reusable
  - If you implement everything in main, you won’t be able to reuse the solution easily in other programs since main is not an explicitly callable member function
How do you start?

- Write down the main() first
  - Write code with which you will test your solvers
  - Compile it and keep it ready
  - This gives you an idea about the interface that your solver should support
- Then implement your problem solver
- Keep your implementation compliable and executable at all times.
  - You are not faced with all the problems in one go
  - The solution architecture will evolve incrementally from externally available interfaces down to detailed implementation structures.
Some Data Structures

- **Stacks**
  - e.g. a stack of books on my table
- **Lists**
  - e.g. a list of students in my class
- **Trees**
  - e.g. a family tree
- **Graphs**
  - e.g. cities in India connected by rail network
Some Algorithms

- Search
  - e.g. search a word in a dictionary
- Sort
  - e.g. rank the students in my class
- Traversal over Graphs
  - e.g. find shortest path between two cities
The Stack (last in first out)

Operations

- `push( )`: pushes an element on the top of stack
- `pop( )`: removes the element at the top of stack
- `empty( )`: returns true if stack empty else returns false
- `full( )`: returns true if stack is full else returns false

(full is implemented on bounded stacks)

Initially stack is empty
Stack - Snapshots

Empty Stack

push ('A');
push ('B');
push ('C');
Boundary Conditions

- Initialize the object with appropriate initial values (e.g. what is the initial value of the variable *top*)
- Take care of boundary conditions when operations are invoked
  - e.g. when push is called: *is stack full?*
  - When pop is called: *is stack empty?*
- *Either throw exceptions or return error codes on unsuccessful operations*
Checking for Matching Parenthesis in expressions

✓ \{ [ 2*a - 2 (b+c) ] * [sin (x+y)] ] \}
✗ \{ [ 2*a - 2 (b+c) } * [sin (x+y)] ]

The Solution?
Checking Matching Parenthesis in expressions using parenthesis Stack

\[ \{ [ 2*a - 2 (b+c) ] * [\sin (x+y)] \} \]

**The Algorithm:**

Stack is initially empty

Scan the expression string from left to right

If a left parenthesis is encountered: push it on the stack

If a right parenthesis is encountered, pop the top of stack and check if the type of popped parenthesis is the same as the type of scanned parenthesis

*failure*: upon mismatch

*success*: if whole string gets scanned without a mismatch
Evaluating postfix expressions using Stack

- Infix expression
  - operand  Operator  operand

- Postfix expression
  - operand  operand  Operator

- Infix  Postfix
- (x+y)  x y +
- (x-y-z)  x y - z -
- (x-y-z)/(u+v)  x y - z - u v + /

- Stack
The Queue: FIFO

Operations

- **insert ( )**: insert an element at the rear end of the queue
- **fetch ( )**: remove the element at the front of the queue
- **empty ( )**: determine whether the queue is empty
- **full ( )**: determine whether the queue is full

Initially, the queue is empty
Implementing Queues
(The circular implementation)

Incrementing the indices:
Modulo Array size

Boundary Conditions
When is the queue empty?
When is it full?
Implementing Queues
(The linked implementation)
Some Examples of Queues

- Process Queues in operating systems
  - ready queues
  - wait queues
- Printer Queues
- Mail queues for incoming and outgoing messages
Arrays: Merge two sorted Arrays

Array A: 1 4 4 20 25 28 50 100 120

Array B: 5 7 7 23 30 35 40 45

Array C: the merged list in sorted order

1 4 4 5 7 7 20 23 25 28 30 35 40 45 50 100 120
Searching through arrays

- Find out the smallest index $i$ such that $A[i] == x$ in an ordered list of elements
- When do you terminate your search, and what index value do you return when the element is not found?

```
<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>23</td>
<td>50</td>
<td>183</td>
<td>187</td>
<td>250</td>
<td>284</td>
<td>299</td>
<td></td>
</tr>
</tbody>
</table>
```

Desired location
Exercise 3: More Stack based Exercises

1. Implement a solution to the matching parenthesis problem using the stack class that you have developed.
2. Implement a solution for postfix expression evaluation

Implement at least one of the above
Exercise 4: Some Array based Problems

1. Implement a circular queue using a bounded array.
2. Implement a function that merges 2 sorted integer arrays
3. Implement binary search on a sorted integer array

Implement all of them.
Exercise 5: More Recursive solutions

- Implement function \textit{factorial} to compute and return the factorial of nonnegative number \( k \) provided as its input argument.
- Implement a recursive function \textit{fibonacci} that computes and returns \( k^{\text{th}} \) Fibonacci number when a nonnegative value \( k \) is provided as its input argument.

\textbf{Implement at least one.}

Also print the total no. of calls made to the function.