Announcements

- 2 TAs assigned (listed on the course web page)
  - Shared with COMP 410-001 section
  - At least one hour of TA office hour available each week day
- My office hour for this week – by appointment only (send me email)
- Make sure you are sitting according to your group # for the next 2 weeks.
- How many of you checked the homework column on the course web-page?
- First exam – 2 weeks from today
- Attendance
Topic Overview

- Abstract Data Types
- Stacks & Queues
  - ADTs
  - Applications
  - Implementations
- O-notation, math review

ABSTRACT DATA TYPES

Abstractions, Data Structures, Implementations
Data Collections

- Data Structures are for collections of data
  - Several objects of the same type

- Examples?
  - Databases:
    - Student list in a course
    - Employee list in an organization
    - Resident list in a community
  - Computational problems:
    - “Tokens” in compilers
  - Resource management:
    - “Processes” in Operating Systems
    - “Customers” in Home Access Networks

Abstractions

- One of the 3 major concepts in the official syllabus:
  - Abstraction
    - Abstractions simplify by hiding the details
      - CS is too complex to know everything
    - Allow us to know about more things
      - e.g., circuits, bits, bytes, Java, data structures, ...
    - As a user, abstractions are helpful
      - But someone has to know the details – you, for data structures
  - Correctness
    - Always a requirement
      - We’ll deal with only through your implementations (pre-conditions, post-conditions, testing)
  - Efficiency (algorithmic complexity)
    - Which choice will be more efficient for large amount of data?
- We’ll focus mostly on abstraction and efficiency
Abstract Data Types (ADTs)

- **Abstract Data Types:**
  - A set of objects together with a set of operations
  - Mathematical abstraction
    - Does not describe how the set of operations is implemented

- **Algorithm:**
  - A high level, language-independent description of a step-by-step process

- **Data Structure:**
  - A specific organization of data and family of algorithms for implementing an ADT (data + operations)

- **Implementation of a data structure or ADT:**
  - A specific implementation in a specific language

Examples in Java

- **Java interfaces** can be thought of as describing an ADT
  - e.g., List, Queue, Set, interfaces...
  - Define what operations can be performed (e.g., for List):
    - `size()` – returns the # of elements in the list
    - `add(e1,index)` – inserts element at specified position
    - `remove(index)` – removes element at specified position
    - `get(index)` – returns element at specified position
    - `set(index,e1)` – replaces element at specified position
  - Interfaces are different from class implementations

- **Java classes** implement the interfaces
  - `ArrayList` and `LinkedList` implement the `List` interface
  - `HashSet` and `TreeSet` implement the `Set` interface
A Word on “Pseudo” code

- In our lectures, we’ll be using (mostly) pseudo-code
  - Not actual Java code, but close enough
    - Do not try to run it as-is
  - Mostly, we will use code “snippets”
    - Not complete definition of a Java class
    - e.g., constructors, error checking, etc will need to be added to every class you write
  - Will be in this font

STACKS & QUEUES

ADT, Applications
Stacks are (last-in-first-out) LIFO lists
- Only the top element is accessible
- Usually drawn vertically

Supports operations:
- push – add an element at the top
- pop – remove an element from the top
- top – return (not remove) the element at the top
- isEmpty – return true if the Stack is empty
- ...

Example Stack:
- push(6)
- push(3)
- push(1)
- push(4)

In-class Problem: Stack Operation

Assume we have a Java class IntegerStack

What is the output of this pseudo-code?
- POSTED ON PIAZZA
Stack Application: Balanced Parenthesis

- How to check if parenthesis are balanced in a mathematical expression?
  - Algorithm:
    - Read expression symbol-by-symbol
    - For each opening parenthesis encountered, push it onto the stack
    - For each closing parenthesis encountered, pop the top element from the stack
    - If parenthesis matched, stack should be empty at the end

- Examples:
  - $(12 * 3 + (42 / 4) + (3 – 5 * (4 / 2))) + 42$
  - $(12 * 3 + (42 / 4) + (3 – 5 * (4 / 2)) + 42$
  - This is exactly how compilers check your code and return errors!

Other Stack Applications

- Computing values of expressions
  - Evaluating a postfix expression
    - e.g.: $5 2 3 + 8 * + 3 +$ evaluates to $48$ ($= (5+(2+3)*8)+3$)
  - Infix-to-prefix conversion
    - e.g.: $a+b*c+(d*e+f)*g$ $\rightarrow abc^*+de^*f+g^*$
  - Read algorithms from textbook (and implement them)!

- Process run-time execution in Operating Systems
Another ADT Example – Queues

- Queues are (first-in-first-out) FIFO lists
  - Usually drawn horizontally

- Supports operations:
  - enqueue – add an element at the rear
  - dequeue – remove an element from the front
  - peek – return (not remove) the element at the front
  - isFull – return true if the queue is full
  - ...

- Example Queue:
  
<table>
<thead>
<tr>
<th>enqueue(6)</th>
<th>enqueue(3)</th>
<th>enqueue(1)</th>
<th>enqueue(4)</th>
</tr>
</thead>
<tbody>
<tr>
<td>6</td>
<td>3</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

Example Queue:

- Front
- Rear

In-class Problem: Queue Operation

- Assume we have a Java class `IntegerQueue`

- What is the output of this pseudo-code?
  
  ```java
  IntegerQueue Q = new IntegerQueue();
  Q.enqueue(10);
  Q.enqueue(5);
  Q.enqueue(14);
  int i = Q.peek();
  int k = Q.dequeue();
  int j = Q.dequeue();
  if (j % i) System.out.println("Hello!");
  if (!Q.isEmpty()) System.out.println("Bye!");
  ```
Queue Applications

- Virtually, every real-life line supposed to be a queue
  - First-come-first-served
- Common data structure for arbitrating resource access
  - Printer queue
  - File server queue
  - Network access queue
- Whole branch of mathematics devoted to studying queues!
  - Queuing Theory
  - Probabilistically computes average wait times, queue lengths, etc

STACK/QUEUE IMPLEMENTATIONS

Using Arrays and Linked Lists
Java In-built Implementations: Stack

- One stack implementation is found in the `java.util.Stack` class

- Example code usage:
  - In-class problem: summarize what this piece of code does
    ```java
    int N = 1000;
    Random r = new Random(10);
    Stack<Double> S = new Stack<Double>();
    for (int i=0; i < N; i++)
        S.push(r.nextDouble());
    ```

Your Stack: Implementation Choices

- What if you want to implement your own Stack?
  - And not use Java’s in-built class

- A stack data structure could use an array, a linked list, or anything that can hold data

- In class, we will do:
  - Stack using an array
  - Queue using a linked list

- For first graded homework, you will do:
  - Stack using a linked list
  - Queue using an array
Your Stack: Using an Array

- Let's write it together:
  - What variables do we need?
  - What methods do we need?
    - What are their arguments?
    - What are their return values?
    - What are their pre- or post-conditions?

- How do the above change if we use generic data types?
  - Instead of a specific one (like double)

- What if we wanted to add more elements than size of array?