## Midterm 1 Topics:

The first midterm (March 5 in class) covers in-class material days 1-12, labs 0-3, and reading weeks 1-6. You may bring a 1 page (front and back), hand-written "cheat-sheet" (created by *you*), but no other notes or resources. (You will not need a calculator.) I have put vocab in blue.

1. Data Structures: know the typical instance variables/methods for each data structure and pros/cons of each one. Review the runtime associated with common methods and know the basics of implementing each one from scratch (except arrays). Be able to use them in an application.
   - Arrays (including 2D arrays)
   - Array Lists
   - Linked Lists (including `Node`s, sentinels, singly/doubly linked)
   - Stacks
   - Queues
2. Java fundamentals (including Java syntax and proper coding style)
   - Types and casting
   - Object-oriented programming (OOP) basics
   - Class structure (instance variables, methods, constructors, getters, setters)
   - Java keywords (e.g. static, final)
   - Primitive types (e.g. `int`, `double`, `boolean`, `char`) vs. Objects (e.g. `String`, `ArrayList`, `Node`)
3. Inheritance and Interfaces
   - Syntax and keywords (`extend`, `implement`)
   - Difference between implementing an interface and extending a class
4. Runtime analysis
   - Relationship between runtime analysis and common code structures (e.g. loops)
   - Big-O notation (both theory and practice), i.e. $O(1), O(n), O(\log n), O(n^2), O(n^3)$, etc.
5. Algorithms (+ their runtimes and relevant data structures)
   - Insertion sort (insert elements one at a time, maintaining sorted order)
   - Stack algorithms: function call stack and arithmetic expressions
   - Using a queue to handle different speed of input/output
6. Misc
   - Generics
   - Exceptions
   - Enums and "fall-through"

**Midterm 1 Practice:**
Also review all slides (they sometimes have exercises to complete) and redo all handouts.

1. Loops, Lists, Arrays, and Generics. Suppose there is a `House` class, It has one private instance variable, an `int`, representing the price of the house. There is a constructor that creates a new `House` with a given initial price. A `House` is greater than another `House` if it has a higher price. CS106 has been tasked with writing two methods in main: one to find the `House` with the minimum price given a list of `House`s, and one to find the index of a `House` with a given price in a given array of `House`s. Student X has written the following code to accomplish these tasks.

```
/** Return the house with the minimum price. */
public static House minHouse(LinkedList<House> list) {
    House minHouse = list.get(0); // start out with the first house
    int i=1;
    while (i < list.size()) {
        if (list.get(i).compareTo(minHouse) < 0) {
            minHouse = list.get(i);
        }
        i++;
    }
    return minHouse;
}

/**
 * Return the index of the house with the given price.
 * If no house with this price, return -1.
 */
public static int getPriceIndex(House[] array, int price) {
    int rightIndex = -1;
    for (int i=0; i < array.length; i++) {
        if (array[i].getPrice() == price) {
            rightIndex = i;
        }
    }
    return rightIndex;
}
```

Do these functions accomplish the desired tasks? Why or why not? If not, correct the method(s). If yes, find a way to make the code faster or more elegant. If you have time, write the `House` class (make sure to include a constructor and all necessary methods).

2. Queues and Runtime. CS106 is given a `Queue` class with the following completely functional methods: `enqueue`, `dequeue`, `isEmpty`, `first`, and `size`. Then the task is to write a method within the `Queue` class that adds all the elements of another `Queue`, `B`, to the end of the given queue (while still maintaining order within `b`). However, this must be accomplished without destroying `B` (note we shouldn't use uppercase, one-letter variable names, but it is easier to write on paper with shorter variables). Student X has written the following code.

```
private void addAll(Queue<E> B) {
    while (!B.isEmpty()) {
        E element = B.remove();
        this.add(element);
        B.add(element);
    }
}
```

What is the error in this code? How could you fix it? How could you fix it if you didn't have the `size` method? What is the runtime of your resulting method?

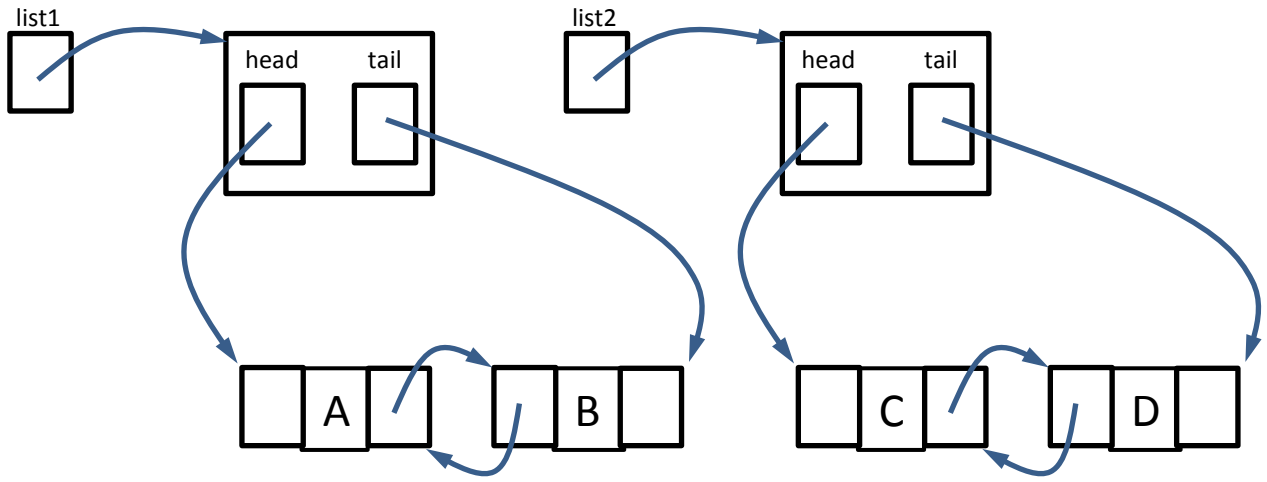
3. Linked Lists. Write a few lines of code (not a method) to split a `LinkedList` of `String`s into two `LinkedList`s of `String`s, with every other element going to a different list. You may assume the original list has an even number of elements.

4. Data Structures

```
import java.util.*;
public class DataStructure<E> {
    private Stack<E> A;
    private Stack<E> B;

    public DataStructure () {
       A = new Stack<E>();
       B = new Stack<E>();
    }

    private void to(int i) {
       for (int j = 0; j < i; j++) {
           B.push(A.pop());
       }
    }

    private void from(int i) {
       for (int j = 0; j < i; j++) {
           A.push(B.pop());
       }
    }

    public E get(int i) {
       E answer;
       to(i);
       answer = A.pop();
       A.push(answer);
       from(i);
       return answer;
    }

    public void set(E x, int i) {
       to(i);
       if (!A.isEmpty()) {
           A.pop();
       }
       A.push(x);
       from(i);
    }
}
```

a.) Which methods of this class may be called by a different class?
b.) Which abstract data type is this class implementing?
c.) How does this implementation compare to the usual implementation in efficiency?
d.) Looking just at the public methods of this class, is there any way for a programmer to tell that it is implemented using stacks? Explain.

5. Linked Lists




(a) Write code to update the links as necessary to remove `B` from `list1`.

(b) Write code to swap the order of the elements in `list1`. Draw a diagram showing the link structure after this operation.

(c) Write code to determine if `list1` is equal to `list2`.