COMP 210: Object-Oriented Programming **May 02, 2018**

Exam 6 (Practice)

Instructions:

- Partial answers will receive partial credit. Make every effort to at least put something for each question. If you don't know the answer, you can get plenty of partial credit by giving a partial answer and explaining where it is wrong or where you got stuck.
- If you believe a problem is incorrectly or incompletely specified, make a reasonable assumption and solve the problem. The assumption should not result in a trivial solution.
- In all cases, clearly state any assumptions you make in your answers.
- Design questions can sometimes have multiple answers. Be sure to give thorough explanations so that I can decide if your answer is a reasonable one I hadn't thought of.

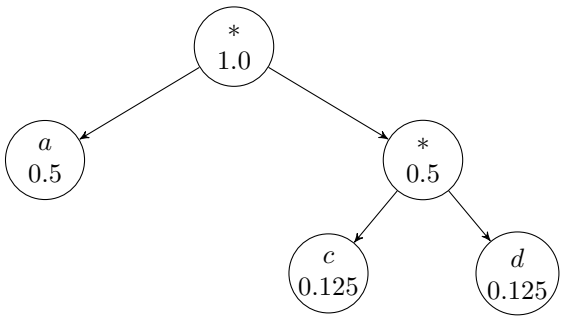
Name	

Question	Description	**Points Possible**	**Grade**
1	Huffman Trees	15	
2	Heaps	15	
3	Heaps	5	
4	Design Principles	20	
5	Design Patterns	30	
Total		85	

1. Create a Huffman tree (using our algorithm from class) for the message

abacadab

You start with four "trees" (leaves): a with frequency $\frac{4}{8} = 0.5$, b with frequency $\frac{2}{7}8 = 0.25$, and c and d with frequency $\frac{1}{8} = 0.125$. Merge the two smallest frequencies: c and d. Then the two smallest are the combination of c and d together with b, so merge those. Then merge with a, yielding:



2. You are given the following array which represents a min-heap. Assume `deleteMin` is called on it. Show the resulting array.

1	2	17	19	4	42	36	25

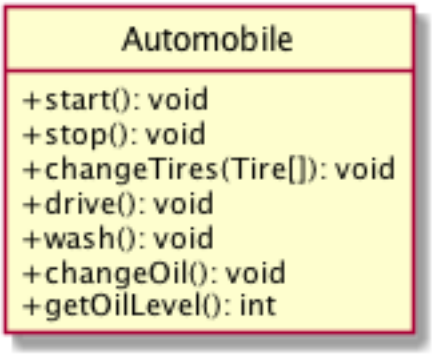
Put the last element (25) into the first slot (replacing 1). Then you need to sift-down: continually swap the node (starting with 25 at the root) with the "dominant" (smaller, in this case) child.

2	4	17	19	25	42	36

3. If I start with an empty min-heap and perform n inserts, what is the *total* worst-case running time (in Big-O)?

$O(n \log n)$

4. You just got a job at Tesla working on autonomous cars. Your boss shows you the following class and says it has a problem, but walks away before you can ask about it.



(a) What SOLID design principle is most violated in this code?

Single Responsibility

(b) How might you fix the problem?

Split the offending methods into other classes that *use* an instance of `Automobile`. For example, `changeTires` and `changeOil` might be in a `Mechanic` class, while `wash` might be in an `Owner` class.

5. Consider the simple class hierarchy concerning pizzas in Listing 1.

```
interface Pizza {
  boolean isBlah();
}

class Crust implements Pizza {
  boolean isBlah() { return true; }
}

class Cheese implements Pizza {
  private Pizza p;
  Cheese(Pizza _p) { p = _p; }
  boolean isBlah() { return p.isBlah(); }
}

class Olive implements Pizza {
  private Pizza p;
  Olive(Pizza _p) { p = _p; }
  boolean isBlah() { return p.isBlah(); }
}

class Anchovy implements Pizza {
  private Pizza p;
  Anchovy(Pizza _p) { p = _p; }
  boolean isBlah() { return false; }
}

class Sausage implements Pizza {
  private Pizza p;
  Sausage(Pizza _p) { p = _p; }
  boolean isBlah() { return false; }
}
```

Listing 1: Simple pizza class hierarchy.

For example, we can make a new anchovy and olive pizza with:

```
Pizza gross = new Anchovy(new Olive(new Cheese(new Crust())));
```

(a) What name makes more sense for `isBlah()`?

`isVegetarian()`

(b) The `isBlah()` method is a bit annoying because it is split between all the subclasses of `Pizza`. What design pattern would allow you to put all the `isBlah` code in one class?

Visitor.

(c) Rewrite the code to use the pattern. You will need new class(es) and new method(s).

This one is a bit tricky because we went through it rather quickly in class. There are several reasonable ways to implement the visitor pattern here. Here is one:

```
interface Pizza { public void accept(PizzaVisitor v); }
class Crust implements Pizza {
        public void accept(PizzaVisitor v) { v.visitCrust(this); }
}

class Cheese implements Pizza {
        private Pizza p;
        Cheese(Pizza _p) { p = _p; }
        public void accept(PizzaVisitor v) {
                v.visitCheese(this);
                p.accept(v);
        }
}

// Code omitted; similar for other Pizza variants (subclasses)

interface PizzaVisitor {
        public void visitCrust(Pizza p);
        public void visitCheese(Pizza p);
        public void visitOlive(Pizza p);
        public void visitAnchovy(Pizza p);
        public void visitSausage(Pizza p);
}

class VeggieVisitor implements PizzaVisitor {
        private boolean result;
        VeggieVisitor() { reset(); }
        public void reset() { result = true; }
        public boolean getResult() { return result; }
        public void visitCrust(Pizza p) { return; }
        public void visitCheese(Pizza p) { return; }
        public void visitOlive(Pizza p) { return; }
        public void visitAnchovy(Pizza p) { result = false; return; }
        public void visitSausage(Pizza p) { result = false; return; }

}

// Wherever you need it:
Pizza gross = new Anchovy(new Olive(new Cheese(new Crust())));
VeggieVisitor v = new VeggieVisitor();
gross.accept(v);
System.out.println("Is Anchovy+Olive+Cheese veggie?: " + v.getResult());
```

An alternative might be to have each `visitX` method to return a boolean which indicates whether or not the "visit" should continue. The `accept` methods must check this before continuing. But this restricts the accept method to returning a boolean to indicate the result. The above solution is more general in that the visitor can accumulate any result type it needs, though it's a bit annoying to have to query the result on the visitor object afterwards.