

Dartmouth College
Computer Science 10, Fall 2014
Midterm Exam

Thursday, October 23, 2014
Professor Drysdale

Print your name: _____

- If you need more space to answer a question than we give you, you may use the backs of pages or you may use additional sheets of paper and attach them to the exam. Make sure that we know where to look for your answer!
- Read each question carefully and make sure that you answer everything asked for. Write legibly so that we can read your solutions. Please do not write anything in red.
- We suggest that for solutions that require you to write Java code, you include comments. They will help your grader understand what you intend, which can help you get partial credit.
- Hand in only what you want graded. We do not want your scrap paper unless it contains solutions you want us to grade.

Question	Value	Score
1	24	
2	20	
3	10	
4	16	
5	15	
6	15	
Total	100	

Question 1 24 points*Short answer***(a)** (4 points)

In Lab 1 you were asked to implement k-means to reduce the number of colors in a picture to k . One way to begin the process was to generate k random colors, and use that color list as the initial color list. When you did that, the final color list usually had many fewer than k colors. Explain what happened, and why it happened so much more often with a random initial color list than when you picked the first k unique colors in the picture as your initial color list.

Solution: What happened is that some of the clusters were empty. An empty cluster happens when a color in the color list is not the closest color to any of the pixels from the picture. It happened more often with random colors because many of the randomly generated colors were far from any color in the picture, so those colors generated empty clusters on the first clustering. Picking k unique colors from the picture guaranteed that at least the first time that you generated clusters every cluster had at least one item in it.

(b) (4 points)

The factory pattern requires a private constructor and a public “factory” method that is the way to actually create an object of the given class. The classes that implemented the `Expression` interface followed this pattern, with each having a `make` or `define` method. What advantage was gained by doing this, as opposed to just having normal constructors?

Solution: Constructors for a class always return objects of that class. A factory method is not restricted that way. If it makes more sense it can return an object of a different class. The `make` methods in the classes that implemented `Expression` used this as a way of simplifying expressions. For example, the `make` method in `Sum` had several cases. It returned a `Constant` object when both `Expressions` to be added were of type `Constant`, a general `Expression` if one of the `Expressions` were the `Constant 0`, and a `Sum` object in other cases. It thus simplified sums of two constants and sums where one of the summands is 0.

(c) (4 points)

The equals method in BinaryTree began:

```
public boolean equals(Object other) {
    if (other instanceof BinaryTree<?>) {
        BinaryTree<E> t = (BinaryTree<E>) other;
```

It would have been much simpler to write it:

```
public boolean equals(BinaryTree<E> t) {
```

and skip all of the silliness of passing an Object and then casting it. What would have gone wrong if we did it the second way?

Solution: The equals method is defined in Object. It is used in a number of methods in Java's built-in classes to test for equality of two objects. An example is the contains method of ArrayList. If we want these methods to work correctly for BinaryTree objects, we have to override this equals method. When we override a method, all of the parameter types must match. The second approach ends up overloading the equals method instead of overriding it. The code for methods like contains in ArrayList all call the version of equals that takes an Object. The first approach overrides this method to supply a tree-specific version. The second approach leaves the default method, but supplies an alternate definition for equals that only is used when a variable of type BinaryTree is supplied at compile time, not when an object of type BinaryTree is referred to by a parameter of type Object at run time.

(d) (4 points)

Exceptions that are not caught are "passed up" the chain of method calls until they reach the main program, and only kill the program if no method in the chain catches the exception. It would be easier to implement exceptions to simply kill the program if the method where the exception occurs does not catch it. What is the advantage of passing the exception back up the call chain instead?

Solution: Exceptions often occur in low-level utility methods. These methods have no knowledge of the overall structure of the program that called them nor what a reasonable way to handle the exception is. By passing the exception back up the call chain there is an opportunity for a method that is in a position to intelligently handle the exception to do so.

(e) (4 points)

A max-heap data structure has two properties: a shape property and a heap-order property. Explain each, and explain why each property is important.

Solution: The shape property says that a heap is a binary tree where every level is full except for perhaps the bottom one, and all of the leaves on the bottom level are as far to the left as possible. That allows the heap to be laid out level-by-level in an array without gaps in a way that child/parent relationships can be determined by arithmetic (basically multiplying and dividing by 2). This means that no pointers are necessary.

The heap-order property says that every child is less than or equal to its parent. This means that the largest item in the heap is at the root, and also allows for $O(\lg n)$ inserts and extract-maximums.

(f) (4 points)

Your `Ellipse` class in Lab 3 had a `drawShape` command that drew an ellipse on a `Graphics` object by calling `drawOval`. Calling `repaint()` when there is an `Ellipse` object in the drawing somehow causes this `drawShape` to be called. Describe the steps in this process. You need not remember all of the names of methods and variables (although some are in the Java reference that we passed out). I will also remind you that the instance variable in `Editor` that referred to the `Drawing` object was `dwg`, and it is visible from the inner class `CanvasPanel`.

Solution: Calling `repaint` causes `paint` to be called. The `paint` method eventually tells the `CanvasPanel` object to re-draw itself. It calls the `CanvasPanel` object's `paintComponent` as part of this process. Within `paintComponent` is a call to the `Drawing` object's `draw` method: `dwg.draw(page)`. This method loops through all of the `Shapes` in the `ArrayList` that is an instance variable of the `Drawing` object, calling `draw` on each. When `draw` is called on an `Ellipse` object the `draw` method in `Shape` is invoked. That method in turn calls `drawShape`.

Question 2 20 points

For a short assignment you appended two lists. For this problem you will splice a section out of a linked list and return it as a new list.

Here is part of the code for the SentinelDLLIterator class. Remember, this version has no current element.

```
public class SentinelDLLIterator<T> implements CS10IteratedList<T> {
    private Element<T> sentinel; // sentinel, serves as head and tail

    private static class Element<T> {
        private T data; // reference to data stored in this element
        private Element<T> next; // reference to next item in list
        private Element<T> previous; // reference to previous item in list

        public Element(T obj) {
            next = previous = null; // no element before or after this one, yet
            data = obj; // OK to copy reference, since obj references an immutable object
        }

        public String toString() {
            return data.toString();
        }
    }

    /**
     * Constructor for an empty, circular, doubly linked list with a sentinel.
     */
    public SentinelDLLIterator() {
        // Allocate the sentinel with a null reference.
        sentinel = new Element<T>(null);
        clear();
    }

    public void clear() {
        // Make the list be empty by having the sentinel point to itself
        // in both directions.
        sentinel.next = sentinel.previous = sentinel;
    }

    // Other methods not shown
}
```

(a) (16 points)

Write a method:

```
public SentinelDLLIterator<T> splice(Element<T> front, Element<T> rear) {
```

It is to remove the sublist between the Element referred to by front and the Element referred to by rear from the list stored in this. It should create a new SentinelDLLIterator list consisting of the sublist and return it. The list stored in this should be re-joined, with the Element before front preceding the Element following rear. So if the original list stored in this consisted of ["a", "b", "c", "d", "e"] and front referred to "b" and rear referred to "d", the call to splice would return the list ["b", "c", "d"] and ["a", "e"] would remain in the list stored in this.

You may assume the following things:

- The list stored in this is not empty.
- front refers to an Element that precedes the Element referred to by rear in the list stored in this or they both refer to the same element.
- Neither front nor rear refers to the sentinel of the list stored in this.

Your code must run in constant time. That is, you must manipulate references rather than deleting individual elements from one list and adding them to the other list. Complete the method body.

```
public SentinelDLLIterator<T> splice(Element<T> front, Element<T> rear) {
```

Solution:

```
    SentinelDLLIterator<T> returnList = new SentinelDLLIterator<T>();
    front.previous.next = rear.next;
    rear.next.previous = front.previous;
    returnList.sentinel.next = front;
    returnList.sentinel.previous = rear;
    rear.next = returnList.sentinel;
    front.previous = returnList.sentinel;
    return returnList;
}
```

(b) (4 points)

Give a set of test data for the `splice` method, consisting of lists and front and rear elements. The first test case is:

List: ["a", "b", "c", "d", "e"]; front: "b"; rear: "d"

Make sure that you test all cases, including boundary cases.

Solution:

1. List: ["a"]; front: "a"; rear: "a"
2. List: ["a", "b", "c", "d", "e"]; front: "a"; rear: "e"
3. List: ["a", "b", "c", "d", "e"]; front: "b"; rear: "e"
4. List: ["a", "b", "c", "d", "e"]; front: "a"; rear: "d"
5. List: ["a", "b"]; front: "a"; rear: "a"
6. List: ["a", "b"]; front: "b"; rear: "b"
7. List: ["a", "b", "c", "d", "e"]; front: "c"; rear: "c"

Question 3 10 points

(a) (5 points)

In class (and in the notes and the book) we saw that repeatedly adding to the end of an `ArrayList` takes $\Theta(1)$ amortized time if the array inside the `ArrayList` doubles in size when it becomes full. We saw that by charging 3 tokens per add operation (where a token can pay for adding an item to an empty spot in the current array or copying a single item from the current array to a new array) we always had enough tokens to pay for the operations.

We also said that the Java implementation does not double the size of the array, but rather increases it by half (so the new array is 1.5 times the size of the current array instead of 2 times the size of the current array). We said that the same argument could be adapted to show that by charging 4 tokens per add operation we will always have enough tokens to pay for the add and the copy operations in this case. Explain how this argument works.

Solution: The important point is that when it is time to create a new array and copy all of the items from the current array to the new array, we always have enough tokens to do so. Consider an array just after it has expanded from size n to size $3n/2$ and the n old items have been copied. It will have $n/2$ empty slots and no tokens accumulated. Each time we fill an empty slot we charge 4 tokens. We use one to pay for the add operation and save 3 of them on the newly added item. When all $n/2$ slots are full we will have accumulated $3n/2$ tokens. That is precisely the number of tokens needed to pay for copying the $3n/2$ items in the array when it becomes full and must be expanded again.

(b) (5 points)

Multiplying two $n \times n$ matrices takes $\Theta(n^3)$ time using the usual algorithm. If it takes .001 second to multiply two 10×10 matrices, about how long will it take to multiply two 1000×1000 matrices?

Solution: The problem size is 100 times as large, so the run time would be $100^3 = 10^6$ times as large. $10^{-3} \cdot 10^6 = 10^3$ or 1000 seconds.

Question 4 16 points

You are hired by the manager of the box office of a theater to help write an automated system for selling tickets. A programmer working for the theater has created the following class to store information about a ticket:

```
public abstract class Ticket
{
    private String eventDate;          // Date that the event is held

    public Ticket(String date) {
        eventDate = date;
    }

    public abstract double getPrice(); // Returns the price of this ticket

    public String toString() {
        return "Date: " + eventDate + "\nPrice: " + getPrice();
    }
}
```

The manager wants you to create additional classes to represent different types of tickets. In particular, he wants you to create:

- StandingRoom: A category of ticket that costs 10 dollars.
- ReservedSeat: A category of ticket that has a row number and a seat number. Reserved seats in rows 1 to 15 cost 30 dollars and other reserved seats cost 20 dollars.
- StudentTicket : A type of ReservedSeat that costs 5 dollars less than what that reserved seat would normally cost.

In all cases the manager wants you to use good program design, using inheritance where appropriate and not re-writing code unnecessarily. Note that each subclass of Ticket should provide or inherit a toString method that returns the following information about the ticket (appropriately labeled): date, price, and where applicable the row and seat numbers.

While in the real world you would of course define constants to represent things like the prices of the various types of tickets and at which row the price changes for reserved seats, for this exam problem just use the values directly in the program. You would probably also use the Java Date class, but here the date is just a string.

(a) (3 points)

Write the class `StandingRoom`. Include a constructor that takes the event date. Include any instance variables or methods that you need. Remember all standing room tickets cost 10 dollars.

Solution:

```
public class StandingRoom extends Ticket
{
    public StandingRoom(String date) {
        super(date);
    }

    public double getPrice() {
        return 10.00;
    }
}
```

(b) (8 points)

Write the class `ReservedSeat`. Include a constructor that takes the event date, the row number, and the column number. Include any instance variables or methods that you need. Remember that reserved seats in rows 1 through 15 cost 30 dollars and other reserved seats cost 20 dollars.

Solution:

```
public class ReservedSeat extends Ticket {

    private int rowNumber, seatNumber;

    public ReservedSeat(String date, int row, int seat) {
        super(date);
        rowNumber = row;
        seatNumber = seat;
    }

    public double getPrice() {
        if rowNumber <= 15
            return 30.00;
        else
            return 20.00;
    }

    public String toString() {
        return super.toString() + "/nRow: " + rowNumber +
            "\nSeat: " + seatNumber;
    }
}
```

(c) (5 points)

Write the class `StudentTicket`. Include a constructor that takes the event date, the row number, and the column number. Include any instance variables or methods that you need. Remember that a student ticket costs 5 dollars less than that seat would normally cost. If the price of a normal reserved seat changes, the price of a student ticket should automatically change.

Solution:

```
public class StudentTicket extends ReservedSeat {  
  
    public StudentTicket(String date, int row, int seat) {  
        super(date, row, seat);  
    }  
  
    public double getPrice() {  
        return super.getPrice() - 5.00;  
    }  
}
```

Question 5 15 points

In sa8 you were asked to add a class `Negate` to the `Expression` hierarchy. For this problem you are to write the class `Square`, which implements `Expression`. One method that you must write is a public `make` method that follows the factory pattern. It should take an `Expression` as its only parameter. If given a `Constant` it should return a `Constant` whose value is the square of the original constant's value. Given any other `Expression` it should return a `Square` object.

Evaluating a `Square` object should return the square of the value of the `Expression` given to the `make` method when the `Square` object was created. Remember that $f(x)^2 = f(x) \cdot f(x)$. If the derivative of $f(x)$ is indicated by $f'(x)$, then we can define the derivative of the square of $f(x)$ as:

$$(f(x)^2)' = 2f(x)f'(x)$$

The following code in `ExpressionDriver`:

```
Variable yVar = define("y", 6.0);
Expression expr = Square.make(minus(Square.make(yVar), constant(4.0)));
System.out.println("The value of the expression:");
System.out.println(expr + " = " + expr.eval());
System.out.println("Its derivative is: " + expr.deriv("y"));
System.out.println("9^2 as simplified in make is: " + Square.make(constant(9.0)));
```

should produce the output:

```
The value of the expression:
((y^2) - 4.0)^2 = 1024.0
Its derivative is: ((2.0 * ((y^2) - 4.0)) * (2.0 * y))
9^2 as simplified in make is: 81.0
```

The outline of the code for class `Square` is given on the next page, with places where you should supply code indicated by something like: `/** Your code here */`

For your convenience the code for the `Expression` interface and the `Constant` class are given on the page following the outline of the code for `Square`. Also, the call to create a new `Product` is:

```
Product.make(expr1, expr2)
```

where `expr1` and `expr2` are of type `Expression`.

Solution:

```
public class Square implements Expression {
    /* Declare instance variable(s) here */

    private Expression expr; //The expression to be squared

    private Square(Expression e) {
        /* Your code here */

        expr = e;
    }

    public double eval() {
        /* Your code here */

        double value = expr.eval();
        return value*value;
    }

    public Expression deriv(String v) {
        /* Your code here */

        return Product.make(Product.make(Constant.define(2.0), expr), expr.deriv(v));
    }

    public String toString() {
        /* Your code here */

        return "(" + expr.toString() + "^2)";
    }

    public static Expression make(Expression e) {
        /* Your code here */

        if(e instanceof Constant) {
            double value = e.eval();
            return Constant.define(value*value);
        }
        else
            return new Square(e);
    }
}
```

Code supplied for your reference. Do not write write or change anything here.

```
public interface Expression {
    /**
     * Evaluates this expression.
     * @return the value of this Expression
     */
    abstract public double eval();

    /**
     * Take the derivative of this expression.
     * @param v the variable with respect to which the derivative is taken
     * @return the derivative of this Expression with respect to the variable v
     */
    abstract public Expression deriv(String v);
}

public class Constant implements Expression {
    private double myValue; // the value of the constant

    /**
     * Constructor
     * @param value the value of this Constant
     */
    private Constant(double value) {
        myValue = value;
    }

    /**
     * Creates a constant
     * @param value the value of the constant
     * @return the constant
     */
    public static Constant define(double value) {
        return new Constant(value);
    }

    /**
     * Evaluates this constant
     * @return the value of this constant
     */
    public double eval() {
        return myValue;
    }
}
```

```
/**
 * Converts this constant to a string
 * @return the string representation of this constant
 */
public String toString() {
    return "" + myValue; // a sneaky way to convert a double to a String
}

/**
 * Take the derivative of this constant
 * @param v the variable with respect to which the derivative is taken (irrelevant)
 * @return the derivative of this constant, which is 0.0
 */
public Expression deriv(String v) {
    return new Constant(0.0); // derivative of a constant is 0
}
}
```

Question 6 15 points

In sa9 you filled in method bodies for the class `StatesAndCities`. The class creates a data structure:

```
Map<String, Set<String>> stateCityMap = new TreeMap<String, Set<String>>();
```

Here the key to the map was a state name and the value was a set of cities that are in the state. In this form it is easy to find which cities are in a state, but hard to find which states contain a city. To make the second question easy to answer you would want a map where the key is a city name and the value is a set of states that contain the city.

Going from the first form to the second form is called inverting a map. So if the original map were:

```
MA: {Boston, Concord}
NH: {Concord, Hanover}
```

then the inverted map would be:

```
Boston: {MA}
Concord: {MA, NH}
Hanover: {NH}
```

You are to write a static method that inverts a map. The parameter is the original map. What is returned is the inverted map. The keys in the original map (e.g. the states) become elements of the value sets in the inverted map, and elements of the value sets in the original map (e.g. the cities) become keys in the inverted map. Complete the method begun on the next page.


```
public static Map<String, Set<String>> invertMap(Map<String, Set<String>> originalMap) {  
  
    // A local variable to accumulate the inverted map, which will be returned.  
    Map<String, Set<String>> invertedMap = new TreeMap<String, Set<String>>();
```

Solution:

```
    for(String key: originalMap.keySet()) {  
        for(String value: originalMap.get(key)) {  
            Set<String> newSet = invertedMap.get(value);  
            if(newSet == null) {  
                newSet = new TreeSet<String>(); // first time value seen, so create new set  
                invertedMap.put(value, newSet);  
            }  
            newSet.add(key);  
        }  
    }  
    return invertedMap;  
}
```