Klover Cloud Assessment

Question 1

I shall use the concept of abstraction and inheritance to describe a circle and a rectangle. I shall first define an abstract class shape which will have abstract functionalities like finding area, perimeter. Then using the principles of inheritance I shall derive specific types of shape circle and rectangle by extending the generalized shape. These specific shapes will implement their own specific area, perimeter finding method.

Please refer here to run the following code.

Shape.java

```java
abstract class Shape {
   private String name;
   public Shape(String name) {
       this.name = name;
   }
   public String getName() {
       return this.name;
   }
   public void setName(String name) {
       this.name = name;
   }
   public abstract void draw();
   public abstract double getArea();
   public abstract double getPerimeter();
}
```

Circle.java

```java
class Circle extends Shape {
   private double radius;
   public Circle(double radius) {
       super("Circle");
       this.radius = radius;
   }
   public void draw() {
       System.out.println("Drawing a
circle...");
   }
   public double getArea() {
       return Math.PI * radius * radius;
   }
   public double getPerimeter() {
       return 2 * Math.PI * radius;
   }
}
```

Rectangle.java

```java
class Rectangle extends Shape {
   private double length;
   private double width;

   public Rectangle(double length, double
width) {
       super("Rectangle");
       this.length = length;
       this.width = width;
   }

   public void draw() {
```

ShapeTest.java

```java
public class ShapeTest {
   public static void shapeDetail(Shape shape){
       shape.draw();
       System.out.println("Shape Name: " +
shape.getName());
       System.out.println("Shape perimeter: " +
shape.getPerimeter());
       System.out.println("Shape area: " +
shape.getArea());
   }
```

```
        System.out.println("Drawing a
rectangle...");
    }
    public double getArea() {
        return length * width;
    }
    public double getPerimeter() {
        return 2 * (length + width);
    }
}
```

```
    public static void main(String[] args) {
      Shape circle = new Circle(5.0);
      Shape rectangle = new Rectangle(5.0, 3.0);
      shapeDetail(circle);
      shapeDetail(rectangle);
    }
}

Output:

Drawing a circle...
Shape Name: Circle
Shape perimeter: 31.41592653589793
Shape area: 78.53981633974483
Drawing a rectangle...
Shape Name: Rectangle
Shape perimeter: 16.0
Shape area: 15.0
```

Runtime Polymorphism: Polymorphism is the ability of an object to take on various forms. For example, the Shape class in the above table has taken two different forms Circle and Rectangle. The `getArea()` and `getPerimeter()` methods are overridden in the child classes of Shape. Call to these overridden methods are resolved dynamically during runtime rather than at compile time. This is known as runtime polymorphism.

Question 2

When a program runs it takes up memory. There are different segments of memory that a program can use. Stack memory refers to that section of memory layout in which local variables of a function are stored. When a function is called, the computer allocates some stack memory for the function. For each new variable declared inside the function, more memory is allocated. When a function returns, the stack memory of the function is deallocated. In contrast to stack memory, heap memory is allocated by the programmer explicitly and it is not deallocated until it is freed. All global variables of a program are stored in the heap memory. Heap memory can be dynamically allocated.

If I need to allocate a very large array (100 MB), I will have to allocate it in heap memory as stack memory is very limited whereas heap memory can provide the maximum memory that an OS can provide.

Question 4

The given recursive approach will not work. Since the tree is heavily left-skewed, the given `traverse()` function will make too many recursive calls which will definitely cross the **maximum recursion depth or maximum stack size** set by the compiler and thereby causing a stack overflow.

An alternative way to overcome the issue with the recursive approach is to use an iterative approach. In this approach, we can use breadth first search (BFS) to iteratively traverse the tree. We initialize an empty queue, push the root node into it. Then for each node we pop from from the queue, we increment the `counter` variable and push the node's children to the queue. The process is followed until the queue is empty. The code for this approach is given below:

Please refer here to run the following code.

```
int modified_traverse(Node *node) {
   if(node == NULL)
       return 0;
   int counter = 0;
   queue<Node *> q;
   q.push(node);
   while(!q.empty()) {
       Node *tmp = q.front();
       q.pop();
       counter++;
       if (tmp->left != NULL)
           q.push(tmp->left);
       if (tmp->right != NULL)
           q.push(tmp->right);
   }
   return counter;
}
```

Programming Questions

BAD_URLS

Please check this: https://github.com/mhhrakib/kc_assesment/blob/main/bad_urls.py

BIG_NUMBERS

Please check this:
https://github.com/mhhrakib/kc_assesment/blob/main/big_numbers.py