Written Question Answers

Question 1

The concept of Object-Oriented Programming stands with the support of four pillars. The are:

a) Abstraction
b) Inheritance
c) Encapsulation
d) Polymorphism

Let's take the examples that were given in the question. A circle and a rectangle both are shapes. Suppose, we have a class called shape. It has some variables and a method named CalculateArea().

```
abstract class Shape
{
        public int height, width;
        public abstract double CalculateArea();

}
```

Here the CalculateArea(); method doesn't have any body. Because this is an abstract method. And we don't know the height and width of the shape to calculate the area.

Now let's create two classes named rectangle and circle in a way that we don't need to create height, width again and we can use the height and width from the shape class. Here comes the concept of Inheritance.

```
Class Rectangle extends Shape
{
        Public Rectangle(int height, int width)
        {
                This.height = height;
                This.width = width;
        }

        Public double CalculateArea()
        {
                Return height*width;
        }

}
```

Here, we are extending the shape class to use it's properties in the rectangle class. This is called Inheritance.

Now creating the Circle class.

```
Class Circle extends Shape
{
	//but the circle doesn't have any height or width it has radius
	//so we will use a private property radius

	Private double radius;

	Public Circle(double radius)
	{
		This.radius = radius;
	}

	Public double CalculateArea()
	{
		Return Math.pow(radius,2)*Math.PI;
	}
}
```

Here we defined radius as private. Private is an access modifier. The radius variable can not be accessed from any other class except the circle class. If we want to pass the value of the radius to other class. We need to use getter and setter for this. This is called Encapsulation.

Now coming to the last pillar which is polymorphism. In the above code example we can see the we modified the calculateArea method as we needed. There are two different types of calculation in two different classes. So, polymorphism means many forms and here we can see the we have two different form of the calculateArea method.

To access these classes we need to create objects of these classes.

So these are the concepts of Object Oriented Programming.

Runtime Polymorphism: Runtime polymorphism occurs in the runtime. Method overriding is a runtime polymorphism. When we use the same method in the child class but change the body of the method, we override the method. Now if a reference is created from parent class but object is created from child class, during compile time the reference is checked, but when we run the program it will execute the method from the object class. We can achieve runtime polymorphism like this:

Suppose we have parent class vehicle and a method move(). We created a child class named Car. Now in the test file:

```
Vehicle v = new Vehicle();
Vehicle c = new Car();

v.move();//it will run the method of vehicle class
c.move(); //it will run the method of car class
```

Question 2

Stack: Stack is a linear data structure. It is a special area of computer's memory that keeps or stores temporary variables produced by functions. Variables are declared, saved and initialized in the stack throughout execution. Stack is a type of temporary memory storage. The memory of the variable will be automatically deleted after the computation process is completed.

Heap: Heap is a hierarchical data structure. The heap is a kind of memory that computer languages use to keep track of global variables. All global variables are kept in heap memory space by default. It has the ability to allocate memory dynamically. The heap is not handles automatically for us and the CPU doesn't have as much control over it. It's more like memory's free-floating area.

For a large size of array(assuming 100mb), I want to allocate memory in the heap for my array. Because, stack is not capable of handling 100MB of size. It has a very limited memory and random access is not possible. The stack will fall outside of the memory area and will give stack overflow errors.
On the other hand, on heap, there is a garbage collection which runs to free the memory used by the object. It has a large block of memory to allocate and it does not have any limit on memory size.

Question 3.

In the given nested loops matrix multiplication is happening. Generally the given code's time complexity is O(n^3) which is huge for millions size of matrices. But there is a way to improve it. (Thanks for the hint :p)
We can use multi-threading to improve it. Instead of using a single core of our CPU to tackle the problem, we can use all or more cores. We can make many threads, each thread analyzing the unique element of matrix multiplication. We can generate the needed number of threads based on the number of cores in our processor. This will work concurrently that means parts of the multiplication will be computed at the same time. Suppose I have a loop to run for 10 times which takes 10ms to execute but I divided it in five different threads each containing 2 part of the loop. So it will take 5ms to execute. Thus we can improve the given problem for very large size of matrix.

Question 4

The given recursive approach is going to work perfectly for smaller size of nodes. But for millions of nodes it will take a lot of time and will be a very slow process. Because the recursive processing is traversing each and every node thus the time complexity of this approach would be O(n), where n = number of nodes.

There is nothing much we can do with the given approach. No matter what we do, it will take O(N) time complexity. But there is another approach I can suggest. If we use the property of a complete binary tree, we can minimize the time. We can measure the height of the tree and add 1 with that as we know total_node = height + 1. Also if we find left height and right height are equal we can use the formula pow(2,height) - 1. The code is as follows:

```
int countNodes(TreeNode* node) {
      TreeNode *cur=node;
      int lh=0,rh=0;
       while(cur!=NULL){  // calculate left height
          lh++;
          cur=cur->left;
       }
       cur=root;
       while(cur!=NULL){ //calculate right height
          rh++;
```

```
    cur=cur->right;
}
if(lh==rh)
    return pow(2,lh)-1;
return 1+countNodes(root->left)+countNodes(root->right);
}
```

Here, first we are calculating the left height by traversing the leftmost part of the tree by increasing the lh variable. Then the right most part. If it finds lh = rh, then it does not need to traverse the whole tree. It can use the formula which is pow(2,lh) - 1. But if it doesn't find them equal, it traverses using the recursive method. In each level of the tree, only one node can possibly trigger the recursion.
In this approach to count the left height it will take logN time and to count the right height(if any) it will take logN time. So the total complexity is (logN * logN).

P.S: Did a lot of research and came up with the above ideas.

References:

1. https://www.geeksforgeeks.org/multiplication-of-matrix-using-threads/
2. https://leetcode.com/problems/count-complete-tree-nodes/discuss/1496246/Do-Your-Children-a-Favor-and-Pass-Down-the-Height-if-You-Know-it-%3A)-Java