Interview Round 4

Puzzel(20*2 Marks, 15 min)

There is a room with a door (closed) and three light bulbs inside the room. Outside the room, there are three switches, connected to the bulbs. You may manipulate the switches as you wish, but once you open the door you can't change them. All bulbs are in working condition and you can open the door only once. Identify each switch with respect to its bulb.

Solution:
Let the bulbs be X, Y, and Z
Turn on switch X for 5 to 10 minutes. Turn it off and turn on switch Y. Open the door and touch the light bulb.
1. the light is on from the bulb, it is Y

Now we will check the other two off bulbs
2. the bulb which is hot, is X
3. the bulb which is cold, is Z

Q-2 Puzzle: Consider a two-player coin game where each Player A and Player B gets the turn one by one. There is a row of even number of coins, and a player on his/her turn can pick a coin from any of the two corners of the row. The player that collects coins with more value wins the game. Develop a strategy for the player making the first turn i.e, Player A, such that he/she never loses the game. Note that the strategy to pick a maximum of two corners may not work. In the following example, the first player, Player A loses the game when he/she uses a strategy to pick a maximum of two corners

```
Initial row:  18 20 15 30 10 14

Player A picks 18, now row of coins is


After first pick:  20 15 30 10 14

Player B picks 20, now row of coins is
```

```
After second pick:  15 30 10 14
Player A picks 15, now row of coins is

After third pick:  30 10 14
Player B picks 30, now row of coins is

After 4th pick:  10 14
Player A picks 14, now row of coins is

Last pick:  10
Player B picks 10, game over.

The total value collected by Player B is more (20 +
30 + 10) compared to first player (18 + 15 + 14).
So the second picker, Player B wins.
```

Solution: The idea is to count the sum of values of all even coins and odd coins, compare the two values. The player that makes the first move can always make sure that the other player is never able to choose an even coin if the sum of even coins is higher. Similarly, he/she can make sure that the other player is never able to choose an odd coin if the sum of odd coins is higher. So here are the steps to a proper algorithm of either winning the game or getting a tie:

Step 1: Count the sum of all the coins in the even places(2nd, 4th, 6th and so on). Let the sum be "EVEN".

Step 2: Count the sum of all the coins in the odd places(1st, 3rd, 5th and so on). Let the sum be "ODD".

Step 3: Compare the value of EVEN and ODD and this is how the first player, here Player A must begin its selection.

if (EVEN > ODD), start choosing from the right-hand corner and select all the even placed coins.

if (EVEN < ODD), start choosing from the left-hand corner and select all the odd placed coins.

if (EVEN == ODD), choosing only the odd-placed or only the even placed coins will throw a tie.

DSA Round(2 question 40 min, 50* 2 marks)

Q-1 Open VS code and Design a stack with operations
1) push() which adds an element to the top of stack.
2) pop() which removes an element from top of stack.
3) findMiddle() which will return middle element of the stack.
4) deleteMiddle() which will delete the middle element.
Push and pop are standard stack operations.

Q-2 Check if a string is a valid shuffle of two distinct strings

React coding challenges(20 marks for code,20 marks for optimization, 30 min):

Q-1 Create Two input one input will used for entering the user and second one for deleting the user, You have to make an list(Sorted) of user add from one first input bar and delete the user by second input bar also implement this functionality by clicking on button

Theory Round(10*5 questions,15 min)

Q-1 Why should we use JSX instead of normal Javascript?
It is faster than normal JavaScript as it performs optimizations while translating to regular JavaScript.
It makes it easier for us to create templates.
Instead of separating the markup and logic in separate files, React uses components for this purpose.
As JSX is an expression, we can use it inside of if statements and for loops, assign it to variables, accept it as arguments, or return it from functions.

Q-2 According to you what may be the reason of changing class attribute in HTML to className in JSX?

The change of class attribute to className:The class in HTML becomes className in JSX. The main reason behind this is that some attribute names in HTML like ‘class‘ are reserved keywords in JavaScript. So, in order to avoid this problem, JSX uses the camel case naming convention for attributes.

Q-3 what are custom attribute in JSX ?

Creation of custom attributes:We can also use custom attributes in JSX. For custom attributes, the names of such attributes should be prefixed by data-* attribute.

```
import React from "react";

import ReactDOM from "react-dom";

const element = (

        <div>

                <h1 className="hello">Hello Geek</h1>

                <h2 data-sampleAttribute="sample">

                        Custom attribute

                </h2>

        </div>

);

ReactDOM.render(element, document.getElementById("root"));
```

Q-4 What are Pure Components in React

ReactJS has provided us with a Pure Component. If we extend a class with Pure Component, there is no need for the shouldComponentUpdate() lifecycle method. ReactJS Pure Component Class compares the current state and props with new props and states to decide whether the React component should re-render itself or not.

In simple words, If the previous value of the state or props and the new value of the state or props are the same, the component will not re-render itself. Pure Components restricts the re-rendering when there is no use for re-rendering of the component. Pure Components are Class Components that extend React.PureComponent.

Pure Components Key Points

Some key points to remember about Pure Components are:

Shallow Comparison:

Pure components perform a shallow comparison of the props and states. If the objects are passed as props or states have the same references, a re-render is prevented.

Performance Optimization:

Pure components can provide performance optimizations by preventing unnecessary re-renders when the data is the same and hasn't been modified.

ShouldComponentUpdate:

Pure components automatically implement the shouldComponentUpdate() method with a shallow prop and state comparison. This method returns false if the props and state haven't changed.

Q-5 What is Hoisting in javascript?

Hoisting in JavaScript is a behavior where variable and function declarations are moved to the top of their containing scope during the compile phase, before the code is executed. This means that regardless of where declarations are within their scope, they are treated as if they are declared at the top. However, it's important to note that only the declarations are hoisted, not the initializations or assignments.
Here's an example to illustrate hoisting with variable declarations:

```
console.log(x); // Output: undefined
var x = 10;
console.log(x); // Output: 10
```

Hoisting in JavaScript is a behavior where variable and function declarations are moved to the top of their containing scope during the compile phase, before the code is executed. This means that regardless of where declarations are within their scope, they are treated as if they are declared at the top. However, it's important to note that only the declarations are hoisted, not the initializations or assignments.
Here's an example to illustrate hoisting with variable declarations:
javascriptCopy code

```
console.log(x); // Output: undefined
var x = 10;
console.log(x); // Output: 10
```

In the above code:
The console.log(x) statement is executed before x is declared. However, it doesn't throw an error. Instead, it logs undefined to the console. This happens because the declaration of x is hoisted to the top, but the assignment (x = 10) remains in place.
Here's another example with function declarations:

```
hello(); // Output: "Hello, world!"

function hello() {
  console.log("Hello, world!");
}
```

In this example:
The hello() function is called before it is declared. However, it still executes without any error because function declarations are hoisted to the top of their scope.
However, hoisting behavior is different for function expressions and arrow functions. Let's see an example:

```
sayHello(); // Output: TypeError: sayHello is not a function

var sayHello = function() {
  console.log("Hello!");
};
```

In this case:
The sayHello() function is called before it's declared, but unlike function declarations, function expressions are not hoisted to the top. As a result, a TypeError occurs because sayHello is undefined at the time of invocation.

To summarize, hoisting is a mechanism in JavaScript that moves declarations (of variables and functions) to the top of their containing scope during the compile phase. Understanding hoisting is important for avoiding unexpected behavior in your JavaScript code.