Hi guys! My name is Yemi, and I would like to use to this medium (get it? Lol) to go through a few Data Structures and Algorithm questions in JavaScript.

Today's discussion will be on Stacks, but before I dive in, I would like to state a couple of assumptions.

Assumptions:

1) I am assuming you are familiar with JavaScript.
2) I am assuming you're familiar with ES6 syntax
3) I am assuming you have some knowledge about big O-notation
4) I am assuming you are familiar with JavaScript Arrays.

Stacks

What is a Stack? Stack is a data structure that contains an ordered list of objects. It follows the Last In, First Out (LIFO) principle. What is the LIFO principle, you say? Well, it is when the last object in a list is removed first. Let's take a look at this diagram.

{insert stacks.jpg}

Creating a Stack

Now that we have an idea of what a Stack is, let's dive into creating one. We are going to use JavaScript arrays to implement a stack class. Then add an item or object to a Stack, remove from a Stack, determine the next item to be removed, check if the Stack is empty, and find the smallest item in a Stack.

{insert stacks1.jpg}

```
class Stack {
// Write methods to do something with a stack
}
```

The above code shows a JavaScript class, Stack. We are going to create methods, Push, Pop, Peek, isEmpty and getMin in the class to perform basic operations on stacks.

But before we do that, we are going to create a constructor method for creating and initializing an array within the class.

```
class Stack {
    constructor () {
        this.data = []; // Creates and initializes an empty array within the class
    }
}
```

Push()

This method adds an item or record to the Stack.

```
Push(record) {
    this.data.push(record); // The Push method inserts a record into the array
 }
```

Pop()

This method removes the last inserted item from the Stack.

```
Pop() {
   return this.data.pop(); // The Pop method removes the last inserted item from the
array
    }
```

Peek()

This method returns the top item in a Stack. The next item to be removed from the Stack.

```
Peek() {
    return this.data[this.data.length-1]; // The Peek method returns the next item to be
removed from the array
    }
```

isEmpty()

This method returns a true or false value if a Stack is empty or not.

```
IsEmpty() {
    return this.data.length === 0 ? true : false; // The isEmpty method checks the
length of an array and returns a true or false value
    }
```

getMin()

Assuming the stack contains numbers, this method returns the smallest value in the Stack.

```
getMin() {
    return Math.min.apply(null, this.data); // The getMin method returns the minimum
value in the array
     }
```

The full implementation of a Stack looks like this.

```
class Stack { // implementing a stack using JavaScript arrays
    constructor () {
        this.data = []; // Creates and initializes an empty array within the class
    }

    Push(record) {
        this.data.push(record); // The push method inserts a record into the array
        }

     Pop() {
       return this.data.pop(); // The Pop method removes the last inserted item from
the array
    }

    Peek() {
        return this.data[this.data.length-1]; // The Peek method returns the next item
to be removed from the array
    }

    isEmpty() {
        return this.data.length===0 ? true : false; // The isEmpty method checks the
length of an array and returns a true or false value
    }

    getMin() {
    return Math.min.apply(null, this.data); // The getMin method returns the minimum
value in the array
     }

}
```

Time Complexity and Stacks

Time Complexity analysis determines how long an algorithm with a given number of inputs (denoted by n), will take to complete a task or an operation. The Big-O Notation is used to represent time complexities.

The **Big-O Notation** – **O(expression),** converts overall steps taken to perform an algorithmic operation into simple algebraic terms.

The insertion and deletion of a Stack take O(1) time to run, also known as constant time. If we were to search a Stack for an item, using the getMin() method, it will take O(n) linear time to run. This means the process will run repeatedly until we find the item.

Example

Consider a Stack called **stackObject**. We are going to create **stackObject** using the Stack class.

```
const stackObject = new Stack(); // creates a new stack object
```

Currently, **stackObject** is an empty stack (empty array). We are going to use the push method to add a couple of items to **stackObject**.

```
stackObject.Push(4);
stackObject.Push(7);
stackObject.Push(6);
stackObject.Push(1);
```

The **stackObject** looks like this.

```
Stack { data: [ 4, 7, 6, 1 ] }
```

If you notice, the last item added to **stackObject** is 1. Following the LIFO principle, this item will be removed first.

```
stackObject.Pop(); // remove the last item inserted
```

After deletion, the **stackObject** looks like this.

```
Stack { data: [ 4, 7, 6 ] }
```

Lets find out the next item to be removed from **stackObject.**

```
stackObject.Peek(); // returns the next value to be removed from stackObject
```

This returns the value **6**. Now, let's check if the **stackObject** is empty.

```
stackObject.isEmpty(); // returns true or false
```

This returns **false**. As expected because we know **stackObject** is not empty. We still have a few items in **stackObject**; let's find the minimum value.

Remember what our stackObject looks like? Well, here it is.

```
Stack { data: [ 4, 7, 6 ] }
```

```
stackObject.getMin(); // returns the minimum value in stackObject
```

This returns the value **4**.

{insert stacks2.jpg}

Conclusion

Good Job guys! We have implemented a Stack using JavaScript arrays. We discussed briefly on time complexity and we worked with an example, **stackObject**.

If you have any questions or concerns on how I implemented a stack or my code, please feel free to message me.

I hope you enjoyed this write up.