Week 2 - Arrays 1 (of 2)

Overview
This week, we explore the *array* data structure: reading and changing elements, as well as adding and removing elements (hence changing the array's length). Before week's end, we will also touch upon associative arrays as well.

Prerequisites
At this point we expect that you can quickly complete the earlier 13 mandatory algorithm challenges. Also, building straightforward Array functions such as min(), max(), sum() and average() should be easy and rapid.

T-Diagrams
Tracking variables with a t-diagram as you write an algorithm is extremely beneficial. Use a t-diagram for challenges this week.

Extra Goodies
Below are some of the Javascript constructs we'll use this week. Remember these basic building blocks!

Concepts
Arrays store multiple values, which are accessed by specifying the *index* (the offset from the front of the array) in square brackets. This *random-access* characteristic makes arrays well-suited for accessing values in a different order than they were added. Arrays are less suitable (but still commonly used) in scenarios with many insertions and removals. Each value must be moved to create space for an insertion (or to fill a vacancy caused by a removal).

Arrays are zero-based: an array's first value is located at index 0. Accordingly, the array attribute *length* means "one more than the last index." As with other interpreted languages, JavaScript arrays are not fixed-length; they automatically grow as new values are set beyond the current length.

Finally, values in a JavaScript array need not all be the same data type. A single array can contain numbers, booleans, strings, objects, functions, etc.

Declaring a new array:
```
    var myArray = [];
    console.log(myArray.length);          //  -> "0"
```

Setting and accessing array values:
```
    myArray[0] = 42;                      //  myArray = [42],                     myArray.length == 1
    console.log(myArray[0]);              //  -> "42"
```

Array.length is determined by the largest index:
```
    myArray[1] = "hello!";                //  myArray == [42, "hello!"],          myArray.length == 2
    myArray[2] = true;                    //  myArray == [42, "hello!", true],    myArray.length == 3
```

Overwriting array values:
```
    myArray[0] = 101;                     //  myArray == [101, "hello!", true]
```

Arrays can be sparsely populated:
```
    myArray[myArray.length + 2] = 0.212;  //  myArray == [101, "hello!", true, undefined, 0.212]
    console.log(myArray.length);          //  -> "5"
```

Shorten an array with the pop() method:
```
    myArray.pop();                        //  myArray == [101, "hello!", true, undefined]
    myArray.pop();                        //  myArray == [101, "hello!", true],  myArray.length == 3
```

Week 2 - Arrays 1 - Monday

This week you will familiarize yourself with basic array manipulation. Here is a list of concepts to consider; some or all will be used in this week's challenges.

for / while loops *array.pop() & push()* *can contain different data types*
if / else statements *arrays grow: arr.length == lastIdx-1* *arrs are objects, passed by reference (ptr)*

<u>**PushFront**</u>

Given an array and an additional value, *insert this value* at the beginning of the array. Do this without using any built-in array methods.

Answer:

<u>**PopFront**</u>

Given an array, *remove and return the value* at the beginning of the array. Do this without using any built-in array methods except pop().

Answer:

<u>**InsertAt**</u>

Given an array, an index, and an additional value, *insert the value into the array* at the given index. Do this without using any built-in array methods.

Answer:

<u>**RemoveAt**</u>

Given an array and an index into the array, *remove and return the array value* at that index. Do this without using any built-in array methods except pop().

Answer:

Tomorrow: u-turns and censorship