Various implementations of stacks

TODO: This section is not ready. Do NOT read.

In this section we will compare showcase some different stack implementations.

Notes

These stack implementations vary from each other in two different ways:

- The way that the values are stored. Some implementations use an array, while others use something akin to a linked list.
- The interface provided. While all implementations provide the same 3 methods, the way they must be called differs from one to the next.

Array-based

These implementations all use an array `arr` to hold the stack values.

This is the implementation we saw previously, here it is reproduced with a slight variation to avoid code duplication:

```
function makeStack() {
    var arr = [];
    function push(el) {
        arr.push(el);
        return null;
    };
    function pop() {
        if (isEmpty()) {
            throw new Error("Attempt to pop from empty stack");
        } else {
            return arr.pop();
        }
    };
    function isEmpty() {
        return arr.length === 0;
    };
    return {
        push: push,
        pop: pop,
        isEmpty: isEmpty
    };
}
```

Notice that we give a name to the object containing the 3 methods.

4. In this question we will construct an alternative to the “stack” implementation we saw in class. This should be close to the linked list approach you may be familiar with from data structures.

We will represent each "entry" with an object: { value: v, next: o } where v is the value and o is the object that had the previous entry, and so on. So we would want the stack to start its life empty, and we will use null to represent that. If we then push the value 5 in the stack, we should end up with { value: 5, next: null }. if we then push 6 we should end up with { value: 6, next: { value: 5, next: null }}, and so on. Each time the top element is in the value property, while the object containing the rest is in the next property. Here is a skeleton for this implementation:

```
function makeStack() {
    var stack = (part i goes here);
    return {
        push: function(v) {
            (part ii goes here);
        },
        pop: function(v) {
            if (... test for empty stack here ...) {
                throw new Error("Attempt to pop from empty stack");
            } else {
                (part iv goes here);
            }
        },
        isEmpty: function() {
            (part iii goes here);
        },
    };
}
```

We will now implement each step. Throughout, the stack variable is meant to hold the current stack, which is really just an object containing the value at the top plus a "link" to the rest.

i. In this part, we look at setting the initial value to the stack variable. Fill in that initial value here:

ii. In this part, we write the body of the function that pushes a new value v in. You will need to set the stack variable to point to a new object that contains the new value and a pointer to the previous stack.

iii. In this part, we write the body of the function that tests to see if the stack is empty. This should be a single return statement.

iv. In this part we write the body of the function that pops an element from the top of the stack. It has an if part, that needs to perform the exact same test as part iii, just negated. You do not need to rewrite that. You need to write what would go into the then clause. You will need to change the stack variable so that it points to the next element, and you need to return the value of the element you just removed. You will probably need 3 lines for this.