Creating a Queue with Limited Data Structures

Learn how to create a queue when the only data structure you have access to is a stack.

Create a Queue Using Two Stacks

This is a fairly common interview question.

Instructions

Write a function that creates a queue using two stacks. Besides these two stacks, you may not use any additional data structures in your implementation. It should have the methods `enqueue`, `dequeue`, and `size`.

Feel free to change the code provided below when you start. It's meant to be a guide.

```
class Queue {
    constructor() {
        this._stack1 = [];
        this._stack2 = [];

        // Your code here
    }

    enqueue(item) {
        // Your code here
    }

    dequeue() {
        // Your code here
    }

    get size() {
        // Your code here
    }
}
```

Succeeded

Solution

```
class Queue {
    constructor() {
        this._enqueueStorage = [];
        this._dequeueStorage = [];
    }

    enqueue(item) {
        this._enqueueStorage.push(item);
    }

    dequeue() {
        if(this._dequeueStorage.length) {
            return this._dequeueStorage.pop();
        }

        if(this._enqueueStorage.length) {
            while(this._enqueueStorage.length) {
                this._dequeueStorage.push(this._enqueueStorage.pop());
            }

            return this._dequeueStorage.pop();
        }

        console.warn('Attempting to dequeue from an empty queue');
        return undefined;
    }

    get size() {
```

```
29          return this._enqueueStorage.length + this._dequeueStorage.length;
30      }
31  }
```

```
Succeeded
```

How it Works

We have two storage stacks, `this._enqueueStorage` and `this._dequeueStorage`. To see how they interact, let's go through an example.

Enqueuing

We want to put 5 elements onto the queue: `1`, `2`, `3`, `4`, and `5`. We enqueue all of these, and they all go into `enqueueStorage`.

```
enqueueStorage: [1, 2, 3, 4, 5]
dequeueStorage: []
```

We now want to dequeue an item. Let's dive into our `dequeue` function.

Dequeuing

We skip the if-statement on line 12 since `dequeueStorage` is empty. We move on to line 16.

Inside that if-statement (line 16), we pop every item off of `enqueueStorage` and put them into `dequeueStorage`. When the while-loop (lines 17 - 19) is finished, `enqueueStorage` is empty and `dequeueStorage` has the five items, but in reverse.

```
enqueueStorage: []
dequeueStorage: [5, 4, 3, 2, 1]
```

On line 21, we pop `dequeueStorage` and return the item, `1`.

```
enqueueStorage: []
dequeueStorage: [5, 4, 3, 2]
```

If we want to dequeue again, we can enter the `dequeue` method once more. This time we go into the first if-statement and pop an item off of `dequeueStorage` and return it.

```
enqueueStorage: []
dequeueStorage: [5, 4, 3]
```

We can keep dequeuing if we like. As long as `dequeueStorage` has items in it, the last item will be popped off and returned to us.

Summary

These steps together make it so our queue functions properly. Enqueuing pushes items onto one stack. Dequeuing pops them from the second stack. When the second stack is empty and we want to dequeue, we empty the first stack onto the second, reversing the order of items.

Enqueue Time

Every enqueue event is a simple push onto a stack. This means that each enqueue has a time complexity of:

O(1).

Dequeue Time

This gets a little more complicated. If we have items in our `dequeueStorage` stack, we simply pop one off and return it, so we have `O(1)` complexity. However, if that stack is empty, we have to transfer every item from `enqueueStorage` to `dequeueStorage` - an `O(n)` operation - and then return an item.

So, for most cases, we have `O(1)`, and for a few rare cases, we have `O(n)`. We can merge these into a single time complexity.

Queue Lifecycle

Let's track the life of three items in our queue. Starting with an empty queue, let's enqueue `1`, `2`, and `3`. This puts them on our first stack.

```
enqueueStorage: [1, 2, 3]
dequeueStorage: []
```

So far, we've performed 3 operations: inserting each of those items onto the stack.

Let's dequeue all three items. First, all items from `enqueueStorage` get transferred to `dequeueStorage`.

```
enqueueStorage: []
dequeueStorage: [1, 2, 3]
```

We'll say that a transference is 1 operation. This is another 3 operations total, bringing us up to 6.

After this, all 3 items are popped off and returned to us. This is another 3 operations, bringing us up to 9.

So, for the insertion and removal of 3 items, we perform 9 operations. Repeating this exercise, we can see that for 4 items, we would perform 12 operations. For 5, we would perform 15. See the pattern?

The entire lifecycle of a single item in a queue **always** requires the same 3 operations: a push onto a stack, transference to another stack, and then a pop from that stack.

Amortized Time

Using this logic, the *average* time complexity of a dequeue operation must be `O(1)`. It's entirely possible that a single dequeue operation can require 100 operations, or even over 1000, depending on the length of the queue.

Averaged over every item, we can still say that the *average* dequeue is an `O(1)` operation. This is known as an amortized constant (https://stackoverflow.com/questions/200384/constant-amortized-time) time complexity.

Amortized time refers to the average time taken per operation. So, our `dequeue` function has an amortized time complexity of

$O(1)$.

Creating a Queue Using One Stack

We're going to go even further and restrict ourselves even more. Recently, Google revealed that they have asked applicants to construct a queue using only a *single stack*.

This might seem impossible, but it's doable, using some trickery.

Feel free to try it yourself.

Hints

- This won't have nice time complexities.

- Think about how you might be able to store information without using an array or an object.

```
class Queue {
    constructor() {
        this._storage = [];

        // Your code here
    }

    enqueue(item) {
        // Your code here
    }

    dequeue() {
        // Your code here
    }

    get size() {
        // Your code here
    }
}
```

Succeeded

Solution

```
class Queue {
    constructor() {
        this._storage = [];
    }

    enqueue(...items) {
        this._storage.push(...items);
```

```
 8      }
 9
10      dequeue() {
11          if(this._storage.length > 1) {
12              const lastItem = this._storage.pop();
13              const firstItem = this.dequeue();
14              this.enqueue(lastItem);
15              return firstItem;
16          } else if (this._storage.length === 1) {
17              return this._storage.pop();
18          } else {
19              console.warn('Attempting to dequeue from an empty queue');
20          }
21      }
22  }
```

Note that in the constructor, we create only a single array. This will be our stack.

The `enqueue()` function passes the items it receives into storage.

How `dequeue()` Works

This recursive method is the fun part. We'll start by discussing the base cases. Lines 16 onward state that if the queue is empty, print a warning and do nothing. If there's a single item, pop it off and return it.

For the cases where there is more than one item present in storage, we can look at lines 11 - 15.

Assume that our storage stack has values `1`, `2`, `3`, in that order, present inside.

```
[1, 2, 3]
```

Since `1` was entered first, when we dequeue, we need to return `1`. We also need to preserve the order of `2` and `3`.

Currently, we're on the first call of this recursive function.

In the First Call

Once we hit line 12, `lastItem` is equal to `3` and our queue has had `3` removed:

```
[1, 2]
```

Since we know more items are present (since this is the case in which there were multiple items in storage), we recursively call the `dequeue()` method.

Entering the Second Call

We enter the function again. Length is now 2, so we enter the same case, lines 11 - 15. Again, we dequeue. In this call, `lastItem` is equal to `2`. Our queue loses another value:

```
[1]
```

We again call the function.

Entering the Third Call

This time, there is only one item, `1`. We pop it off and return it, ending up back in the second call. Our queue is empty.

```
[]
```

Back to the Second Call

We go back to the 2nd call, now on line 14. `lastItem` is `2` and `firstItem` is `1`.

On line 14, we enqueue `lastItem`, or `2`.

```
[2]
```

We return `1` and go back to the first call.

Back to the First Call

We're back at the beginning. `lastItem` is `3` and `firstItem` is `1`. We enqueue `3`:

```
[2, 3]
```

and return `1`.

Dequeue Time

The time complexity of `dequeue()` is:

O(n)

because we need to pop and re-insert every item except one.

Dequeue Space

The space complexity of `dequeue()` is:

O(n)

because we need to call the function one more time for every item.

Conclusion

Think about how challenging this problem is. On the surface, it truly seems impossible. How are we supposed to return the first item in a stack, but also preserve the remaining items, without any additional data structures? Don't we at least need a second stack?

The Second Stack

A recursive function gives us access to that second stack. Rather than creating it ourselves in our function, we're using the JavaScript engine's *call stack*.

The Algorithm

Each recursive function call independently maintains its own piece of data in one single variable. We simply call the function over and over, each time giving it an item off of the stack.

Once we get to the end of the stack we're at the very last function call. This one takes the last item out of the stack and returns it to the 2nd-to-last function call.

The 2nd to last call puts the value it popped *back onto the stack*. It passes the value it received upwards.

All previous function calls put back the value they popped off. Afterwards, they all return the same value up the chain, the one they received from the previous call.

The Call Stack

The call stack's normal purpose is to maintain a record of the order in which functions are called. It also maintains function state, including all variables, until the functions finish their work.

We're highjacking the call stack to store values we want, and then getting them back out when we need them. It's essentially an abuse of the call stack, but it's a beautiful hack that works.

← Back
Creating a Queue with O(1) Operations

Next →
Linked List Cycles

✓ Mark as Completed

Report an Issue