Data Structures With JavaScript: Stack and Queue

Two of the most commonly used data structures in web development are stacks and queues. Many users of the Internet, including web developers, are unaware of this amazing fact. If you are one of these developers, then prepare yourself for two enlightening examples: the 'undo' operation of a text editor uses a stack to organize data; the event-loop of a web browser, which handles events (clicks, hoovers, etc.), uses a queue to process data.

Now pause for a moment and imagine how many times we, as both a user and developer, use stacks and queues. That is amazing, right? Due to their ubiquity and similarity in design, I have decided to use them to introduce you to data structures.

A Stack

In computer science, a stack is a linear data structure. If this statement holds marginal value to you, as it originally did with me, consider this alternative: A stack organizes data into sequential order.

This sequential order is commonly described as a stack of dishes at a cafeteria. When a plate is added to a stack of dishes, the plate retains the order of when it was added; moreover, when a plate is added, it is pushed towards the bottom of a stack. Every time we add a new plate, the plate is pushed towards the bottom of the stack, but it also represents the top of the stack of plates.

This process of adding plates will retain the sequential order of when each plate was added into the stack. Removing plates from a stack will also retain the sequential order of all plates. If a plate is removed from the top of a stack, every other plate in the stack will still retain the correct order in the stack. What I am describing, possibly with too many words, is how plates are added and removed at most cafeterias!

To provide a more technical example of a stack, let us recall the 'undo' operation of a text editor. Every time text is added to a text editor, this text is pushed into a stack. The first addition to the text editor represents the bottom of the stack; the most recent change represents the top of the stack. If a user wants to undo the most recent change, the top of the stack is removed. This process can be repeated until there are no more additions to the stack, which is a blank file!

Operations of a Stack

Since we now have a conceptual model of a stack, let us define the two operations of a stack:

- `push(data)` adds data.
- `pop()` removes the most recently added data.

Implementation of a Stack

Now let us write the code for a stack!

Properties of a Stack

For our implementation, we will create a constructor named `Stack`. Each instance of `Stack` will have two properties: `_size` and `_storage`.

```
function Stack() {
    this._size = 0;
    this._storage = {};
}
```

`this._storage` enables each instance of `Stack` to have its own container for storing data; `this._size` reflects the number of times data was pushed to the current version of a `Stack`. If a new instance of `Stack` is created and data is pushed into its storage, then `this._size` will increase to 1. If data is pushed, again, into the stack, `this._size` will increase to 2. If data is removed from the stack, then `this._size` will decrease to 1.

Methods of a Stack

We need to define methods that can add (push) and remove (pop) data from a stack. Let's start with pushing data.

Method 1 of 2: `push(data)`

(This method can be shared among all instances of `Stack`, so we'll add it to the prototype of `Stack`.)

We have two requirements for this method:

1. Every time we add data, we want to increment the size of our stack.
2. Every time we add data, we want to retain the order in which it was added.

```
Stack.prototype.push = function (data) {
    // increases the size of our storage
    var size = this._size++;
    // assigns size as a key of storage
    // assigns data as the value of this key
    this._storage[size] = data;
};
```

Our implementation of `push(data)` includes the following logic. Declare a variable named `size` and assign it the value of `this._size++`. Assign `size` as a key of `this._storage`. And assign `data` as the value of a corresponding key.

If our stack invoked `push(data)` five times, then the size of our stack would be 5. The first push to the stack would assign that data a key of 1 in `this._storage`. The fifth invocation of `push(data)` would assign that data a key of 5 in `this._storage`. We've just assigned order to our data!

Method 2 of 2: `pop()`

We can now push data to a stack; the next logical step is popping (removing) data from a stack. Popping data from a stack is not simply removing data; it is removing only the most recently added data.

Here are our goals for this method:

1. Use a stack's current size to get the most recently added data.
2. Delete the most recently added data.
3. Decrement `_this._size` by one.
4. Return the most recently deleted data.

```
Stack.prototype.pop = function () {
    var size = this._size,
        deletedData;
    deletedData = this._storage[size];
    delete this._storage[size];
    this.size--;
    return deletedData;
};
```

`pop()` meets each of our four goals. First, we declare two variables: `size` is initialized to the size of a stack; `deletedData` is assigned to the data most recently added to a stack. Second, we delete the key-value pair of our most recently added data. Third, we decrement the size of a stack by 1. Fourth, we return the data that was removed from the stack.

If we test our current implementation of `pop()`, we find that it works for the following use-case. If we `push(data)` data to a stack, the size of the stack increments by one. If we `pop()` data from our stack, the size of our stack decrements by one.

A problem arises, however, when we reverse the order of invocation. Consider the following scenario: we invoke `pop()` and then `push(data)`. The size of our stack changes to -1 and then to 0. But the correct size of our stack is 1!

To handle this use case, we will add an `if` statement to `pop()`.

```
Stack.prototype.pop = function () {
    var size = this._size,
        deletedData;
    if (size) {
        deletedData = this._storage[size];
        delete this._storage[size];
        this._size--;
        return deletedData;
    }
};
```

```
11
12
13
```

With the addition of our `if` statement, the body of our code is executed only when there is data in our storage.

Complete Implementation of a Stack

Our implementation of `Stack` is complete. Regardless of the order in which we invoke either of our methods, our code works! Here is the final version of our code:

```
01
02
03  function Stack() {
04      this._size = 0;
05      this._storage = {};
06  }
07  Stack.prototype.push = function(data) {
08      var size = ++this._size;
09      this._storage[size] = data;
10  };
11  Stack.prototype.pop = function() {
12      var size = this._size,
13          deletedData;
14      if (size) {
15          deletedData = this._storage[size];
16          delete this._storage[size];
17          this._size--;
18          return deletedData;
19      }
20  };
21
22
23
```

From Stack to Queue

A stack is useful when we want to add data in sequential order and remove data. Based on its definition, a stack can remove only the most recently added data. What happens if we want to remove the oldest data? We want to use a data structure named queue.

A Queue

Similar to a stack, a queue is a linear data structure. Unlike a stack, a queue deletes only the oldest added data.

To help you conceptualize how this would work, let's take a moment to use an analogy. Imagine a queue being very similar to the ticketing system of a deli. Each customer takes a ticket and is served when their number is called. The customer who takes the first ticket should be served first.

Let's further imagine that this ticket has the number "one" displayed on it. The next ticket has the number "two" displayed on it. The customer who takes the second ticket will be served second. (If our ticketing system operated like a stack, the customer who entered the stack first would be the last to be served!)

A more practical example of a queue is the event-loop of a web browser. As different events are being triggered, such as the click of a button, they are added to an event-loop's queue and handled in the order they entered the queue.

Operations of a Queue

Since we now have a conceptual model of a queue, let us define its operations. As you will notice, the operations of a queue are very similar to a stack. The difference lies in where data is removed.

- `enqueue(data)` adds data to a queue.
- `dequeue` removes the oldest added data to a queue.

Implementation of a Queue

Now let us write the code for a queue!

Properties of a Queue

For our implementation, we will create a constructor named `Queue`. We will then add three properties: `_oldestIndex`, `_newestIndex`, and `_storage`. The need for both `_oldestIndex` and `_newestIndex` will become clearer during the next section.

```
function Queue() {
    this._oldestIndex = 1;
    this._newestIndex = 1;
    this._storage = {};
}
```

Methods of a Queue

We will now create the three methods shared amongst all instances of a queue: `size()`, `enqueue(data)`, and `dequeue(data)`. I will outline the objectives for each method, reveal the code for each method, and then explain the code for each method.

Method 1 of 3: `size()`

We have two objectives for this method:

1. Return the correct size for a queue.
2. Retain the correct range of keys for a queue.

```
Queue.prototype.size = function () {
```

```
2         return this._newestIndex - this._oldestIndex;
3     };
```

Implementing `size()` might appear trivial, but you will quickly find that to be untrue. To understand why, we must quickly revisit how `size` was implemented for a stack.

Using our conceptual model of a stack, let's imagine that we push five plates onto a stack. The size of our stack is five and each plate has a number associated with it from one (first added plate) to five (last added plate). If we remove three plates, then we have two plates. We can simply subtract three from five to get the correct size, which is two. Here's the most important point about the size of a stack: The current size represents the correct key associated with the plate at top of the stack (2) and the other plate in the stack (1). In other words, the range of keys is always from the current size to 1.

Now, let's apply this implementation of stack's `size` to our queue. Imagine that five customers take a ticket from our ticketing system. The first customer has a ticket displaying the number 1 and the fifth customer has a ticket displaying the number 5. With a queue, the customer with the first ticket is served first.

Let's now imagine that the first customer is served and that this ticket is removed from the queue. Similar to a stack, we can get the correct size of our queue by subtracting 1 from 5. Our queue currently has four unserved tickets. Now, this is where a problem arises: the size no longer represents the correct ticket numbers. If we simply subtracted one from five, we would have a size of 4. We cannot use 4 to determine the current range of remaining tickets in the queue. Do we have tickets in the queue with the numbers from 1 to 4 or from 2 to 5? The answer is unclear.

This is the benefit of having the following two properties in a queue: `_oldestIndex` and `_newestIndex`. All of this may seem confusing—I'm still occasionally confused. What helps me rationalize everything is the following example I've developed.

Imagine that our deli has two ticketing systems:

1. `_newestIndex` represents a ticket from a customer ticketing system.
2. `_oldestIndex` represents a ticket from an employee ticketing system.

Here's the hardest concept to grasp in regards to having two ticketing systems: When the numbers in both systems are identical, every customer in the queue has been addressed and the queue is empty. We will use the following scenario to reinforce this logic:

1. A customer takes a ticket. The customer's ticket number, which is retrieved from `_newestIndex`, is 1. The next ticket available in the customer ticket system is 2.
2. An employee does not take a ticket, and the current ticket in the employee ticket system is 1.
3. We take the current ticket number in the customer system (2) and subtract the number in the employee system (1) to get the number 1. The number 1 represents the number of tickets still in the queue that have not been removed.
4. The employee takes a ticket from their ticketing system. This ticket represents the customer ticket being served. The ticket that was served is retrieved from `_oldestIndex`, which displays the number 1.

5. We repeat step 4, and now the difference is zero—there are no more tickets in the queue!

We now have a property (`_newestIndex`) that can tell us the largest number (key) assigned in the queue and a property (`_oldestIndex`) that can tell us the oldest index number (key) in the queue.

We have adequately explored `size()` , so let's now move to `enqueue(data)` .

Method 2 of 3: `enqueue(data)`

For `enqueue` , we have two objectives:

1. Use `_newestIndex` as a key of `this._storage` and use any data being added as the value of that key.
2. Increment the value of `_newestIndex` by 1.

Based on these two objectives, we will create the following implementation of `enqueue(data)` :

```
Queue.prototype.enqueue = function (data) {
        this._storage[this._newestIndex] = data;
        this._newestIndex++;
};
```

The body of this method contains two lines of code. On the first line, we use `this._newestIndex` to create a new key for `this._storage` and assign `data` to it. `this._newestIndex` always starts at 1. On the second line of code, we increment `this._newestIndex` by 1, which updates its value to 2.

That's all the code we need for `enqueue(data)` . Let's now implement `dequeue()` .

Method 3 of 3: `dequeue()`

Here are the objectives for this method:

1. Remove the oldest data in a queue.
2. Increment `_oldestIndex` by one.

```

Queue.prototype.dequeue = function () {
        var oldestIndex = this._oldestIndex,
            deletedData = this._storage[oldestIndex];
        delete this._storage[oldestIndex];
        this._oldestIndex++;
        return deletedData;
};

```

In the body of `dequeue()` , we declare two variables. The first variable, `oldestIndex` , is assigned a queue's current value for `this._oldestIndex` . The second variable, `deletedData` , is assigned the value contained in `this._storage[oldestIndex]` .

Next, we delete the oldest index in the queue. After it is deleted, we increment `this._oldestIndex` by 1. Finally, we return the data we just deleted.

Similar to the problem in our first implementation of `pop()` with a stack, our implementation of `dequeue()` does not handle situations where data is removed before any data is added. We need to create a conditional to handle this use case.

```
Queue.prototype.dequeue = function () {
    var oldestIndex = this._oldestIndex,
        newestIndex = this._newestIndex,
        deletedData;
    if (oldestIndex !== newestIndex) {
        deletedData = this._storage[oldestIndex];
        delete this._storage[oldestIndex];
        this._oldestIndex++;
        return deletedData;
    }
};
```

Whenever the values of `oldestIndex` and `newestIndex` are not equal, then we execute the logic we had before.

Complete Implementation of a Queue

Our implementation of a queue is complete. Let's view the entire code.

```
function Queue() {
    this._oldestIndex = 1;
    this._newestIndex = 1;
    this._storage = {};
}
Queue.prototype.size = function () {
    return this._newestIndex - this._oldestIndex;
};
Queue.prototype.enqueue = function (data) {
    this._storage[ this._newestIndex] = data;
    this._newestIndex++;
};
Queue.prototype.dequeue = function () {
    var oldestIndex = this._oldestIndex,
        newestIndex = this._newestIndex,
        deletedData;
    if (oldestIndex !== newestIndex) {
        deletedData = this._storage[oldestIndex];
        delete this._storage[oldestIndex];
```

```
21            this._oldestIndex++;
22            return deletedData;
23        }
24    };
25
26
27
28
```

Conclusion

In this article, we've explored two linear data structures: stacks and queues. A stack stores data in sequential order and removes the most recently added data; a queue stores data in sequential order but removes the oldest added data.

If the implementation of these data structures seems trivial, remind yourself of the purpose of data structures. They aren't designed to be overly complicated; they are designed to help us organize data. In this context, if you find yourself with data that needs to be organized in sequential order, consider using a stack or queue.