Queues

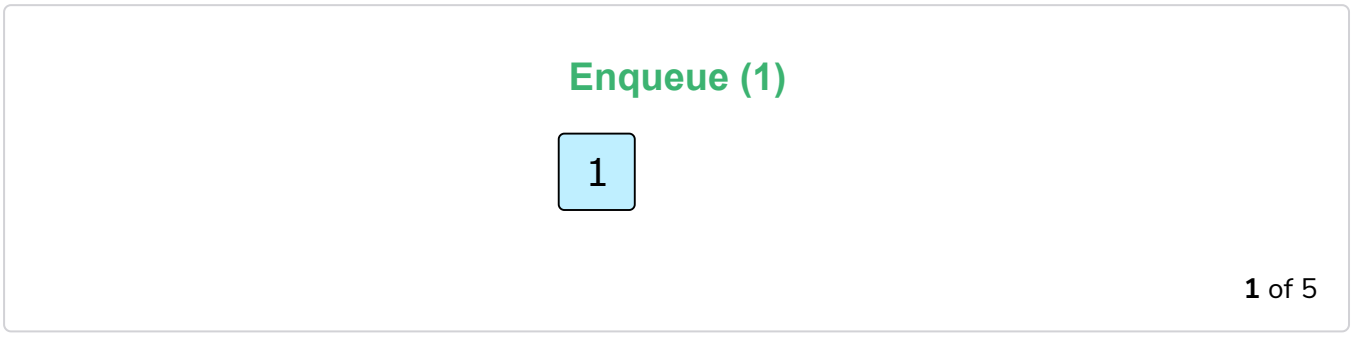
Queue is a FIFO data structure. In this part, we look at Queues and how can they be implemented in Javascript

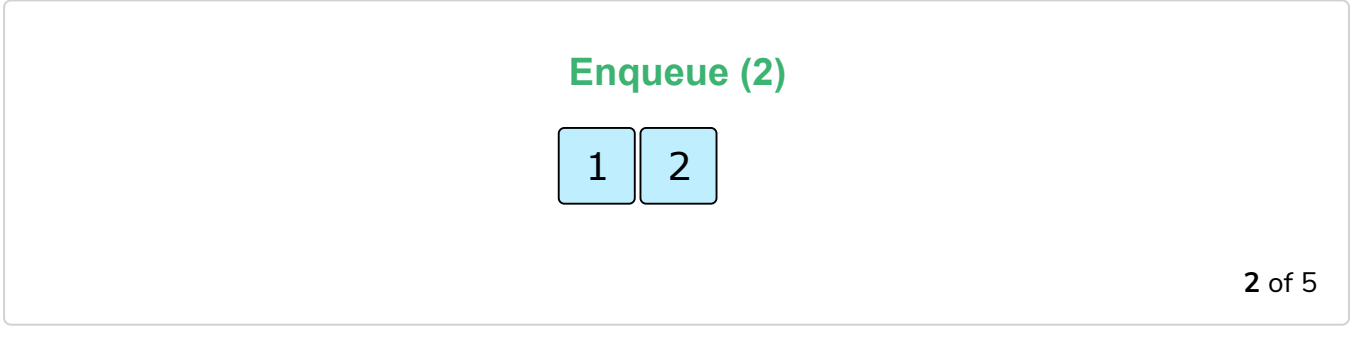
Introduction

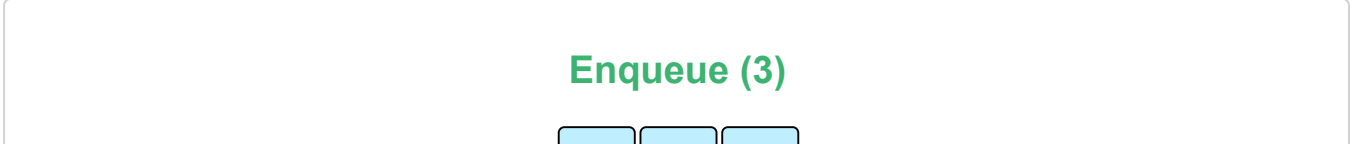
A queue is a linear collection where items are inserted at the end and are removed from the front. Hence queue is a ***FIFO (first-in first-out)*** data structure. It supports three basic operations: Insert and Remove.

1. Inserting into the queue is called ***Enqueue***.
2. Extracting from the queue is called ***Dequeue***.
3. Look at the top element in the queue (the one that will be removed on Dequeue). It is called ***Peek***.

Queue is an abstraction of the behavior that is very common in our daily lives e.g. drive through of a Coffee Shop. It helps process requests or tasks in the order that they arrive. Here's a quick animation on how queue works.

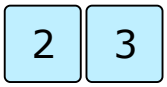






3 of 5

Dequeue() = 1



4 of 5

Dequeue() = 2

5 of 5

> Queue is often compared and studied along with Stack which is a LIFO data structure.

Queue Implementation

We'll implement the Queue data structure using the following object.

```
function Queue() {
  this._head = 0;
  this._data = [];
}
```

We are going to store the queue items in an array.

- *data* contains the items in the queue. At every enqueue, we push the new item into this array.
- *head* is the index of the top or head of the queue. At every dequeue, we return the element pointed by the head and then increment the head to point to the next item.

Let's look at the code for Enqueue and Dequeue below.

Enqueue elements in the Queue

Here's how the Enqueue function would look.

```
Queue.prototype.enqueue = function(data) {
  this._data.push(data);
};
```

Dequeue items from the queue

Here's how a (*rather simplistic*) Dequeue implementation would look. If you have figured out an issue with this method along the lines of excessive memory usage stay with us it will be discussed later on.

```
Queue.prototype.dequeue = function() {
  if (this._head < 0 ||
      this._head >= this._data.length) {
    return null;
  }

  var dequeuedItem = this._data[this._head];
  this._head++;

  return dequeuedItem;
};
```

Peek at the queue

Here's how peek function would look. It's very similar to the Dequeue operation except the head doesn't move to the next item.

```
Queue.prototype.peek = function() {
  if (this._head < 0 ||
      this._head >= this._data.length) {
    return null;
  }

  return this._data[this._head];
};
```

Let's look at the complete implementation of queue.

> Run the following code to see Queue in action

Run the following code to see Queue in action.

JavaScript

HTML

CSS (SCSS)

```
var queue = new Queue();
console.log("Enqueue 100");
queue.enqueue(100);

console.log("Enqueue 200");
queue.enqueue(200);

console.log('Dequeue: ' + queue.dequeue());
console.log('Dequeue: ' + queue.dequeue());
console.log('Dequeue: ' + queue.dequeue());
```

Console

Clear

```
Enqueue 100
Enqueue 200
Dequeue: 100
Dequeue: 200
Dequeue: null
```

Queues Basics

1 Is Queue a LIFO data structure?

2 Which operation decreases the length of the queue?

Check Answers

Exercise 1

As the first exercise, let's implement the *size* function. It returns the size of the queue.

> Some test cases are failing. Fix the size method and click Run button.

JavaScript

HTML

CSS (SCSS)

```
Queue.prototype.size = function() {
  // TODO:
  // IMPLEMENT THIS
  return undefined;
}

// This is the function that runs test cases.
runEvaluation();
```

Console

Clear

```
*** There is some bug lurking there. See failed test cases ***
Here are the tests that ran:
Test case FAILED for queue.size(). Result: undefined. Expected: 0
Test case FAILED for queue.size(). Result: undefined. Expected: 1
```

```
Test case FAILED for queue.size(). Result: undefined. Expected: 0
```

Exercise 2

As a next exercise, let's implement the *isEmpty* function. It returns true when queue has no items and returns false otherwise.

> Some test cases are failing. Fix the isEmpty method so that they can pass.

JavaScript

HTML

CSS (SCSS)

```
Queue.prototype.isEmpty = function() {
  // TODO:
  // IMPLEMENT THIS
  return undefined;
}

// This is the function that runs test cases.
runEvaluation();
```

Console

Clear

```
*** There is some bug lurking there. See failed test cases ***
Here are the tests that ran:
Test case FAILED for queue.size(). Result: undefined. Expected: true
Test case FAILED for queue.size(). Result: undefined. Expected: false
Test case FAILED for queue.size(). Result: undefined. Expected: true
```

Our dequeue implementation leaks memory

For learning purposes, we used a simpler version of Dequeue that leaks memory. Can you figure out how?

When we dequeue, we never release the items from the array. With every dequeue, we move the head forward but the array still holds the reference to the item. It means, all the elements in the array at indexes less than the head are garbage.

To fix this, we cleanup the array. We don't have to slow down each dequeue as deleting from the front of the array is a slow operation. We only do it after a certain threshold. e.g. we can say that we only cleaup the array when the garbage is 10 items or 100 items. Let's look at the code for a dequeue method where we cleanup periodically.

```
Queue.prototype.dequeue = function() {
  if (this._head < 0 ||
      this._head >= this._data.length) {
    return null;
  }

  var dequeuedItem = this._data[this._head];
  this._head++;

  if (this._head === 100) {
    // We have 100 items in garbage
    // Remove items at indexes 0 to 99.
    this._data.splice(0, 100);

    // Reset the head
    this._head = 0;
  }

  return dequeuedItem;
};
```

Does Javascript has built-in support for Queues

It does. Just like stacks, queues can be implemented using Javascript's array. Here's how.

Enqueue can be implemented by using Array.push

Dequeue can be implemented by using Array.shift which removes the first element from the array.

However, you can see that calling Shift at each dequeue is slow as it mutates the array. Hence, if you are writing performant code, the array we implemented is better as it has periodic/lazy cleanup instead of an aggressive cleanup.

Summary

- A queue is a first-in-first-out (FIFO) data structure
- It supports Enqueue for insertion and Dequeue for extraction of items.
- Peek operation return the top most item in the queue.
- Javascript array supports Enqueue (Array.push) and Dequeue (Array.shift)
- Implementing dequeue using Array.shift can be slow.
- Dequeue operation should do periodic cleanup to save memory.