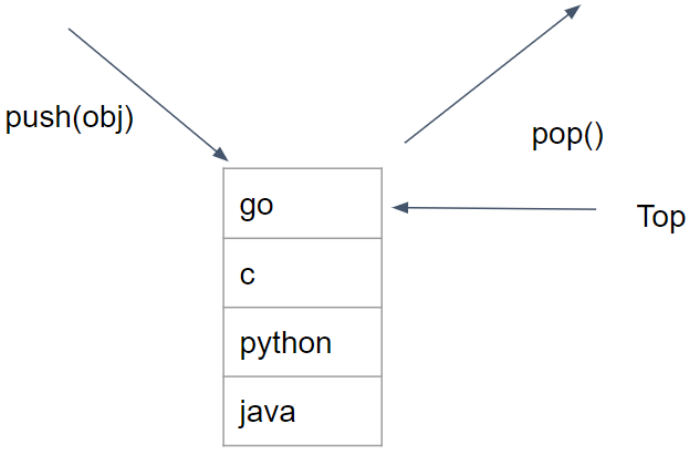
Agenda :

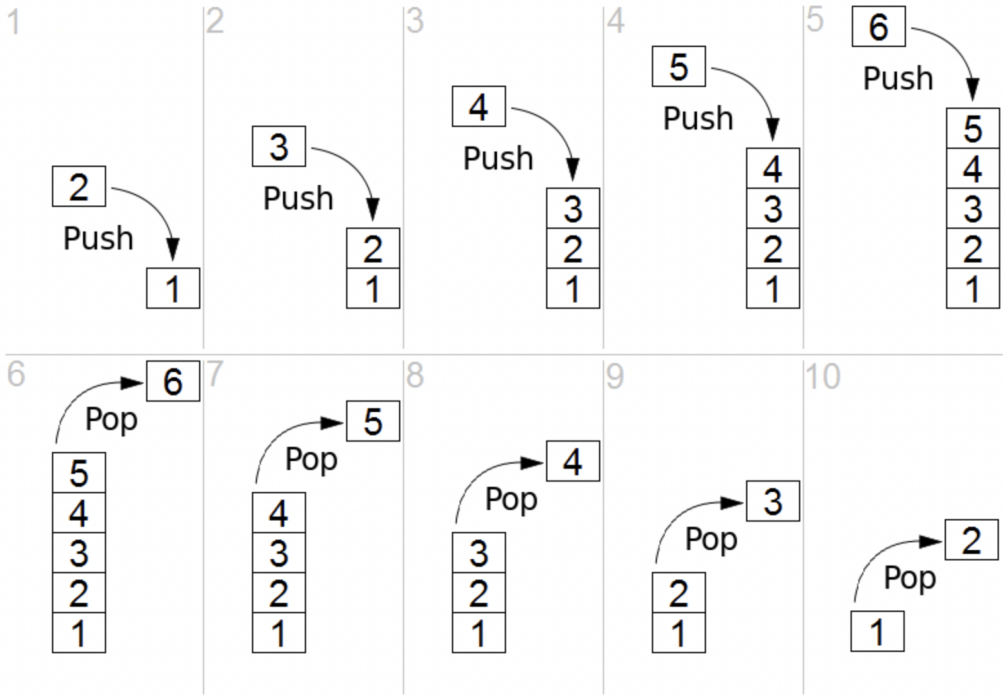
- Intro to Stacks
- Intro to Queues
- BIG O of Queues

What is a Stack ?

- A stack is a list with the restriction that insertions and deletions can be performed in only one position, namely, the end of the list, called the top.
- The operations: push (insert) and pop (delete)



Stack Representation :



Example: Implement Stack

```
// program to implement stack data structure
class Stack {
    constructor() {
```

```
        this.items = [];
    }

    // add element to the stack
    add(element) {
        return this.items.push(element);
    }

    // remove element from the stack
    remove() {
        if(this.items.length > 0) {
            return this.items.pop();
        }
    }

    // view the last element
    peek() {
        return this.items[this.items.length - 1];
    }

    // check if the stack is empty
    isEmpty(){
       return this.items.length == 0;
    }

    // the size of the stack
    size(){
        return this.items.length;
    }

    // empty the stack
    clear(){
        this.items = [];
    }
}

let stack = new Stack();
stack.add(1);
stack.add(2);
stack.add(4);
stack.add(8);
console.log(stack.items);

stack.remove();
console.log(stack.items);

console.log(stack.peek());

console.log(stack.isEmpty());

console.log(stack.size());

stack.clear();
console.log(stack.items);
```

Output

```
[1, 2, 4, 8]
[1, 2, 4]
4
false
3
[]
```

In the above program, the `Stack` class is created to implement the stack data structure. The class methoc like `add()` , `remove()` , `peek()` , `isEmpty()` , `size()` , `clear()` are implemented.

An object stack is created using a `new` operator and various methods are accessed through the object.

- Here, initially is an empty array. this.items
- The `push()` method adds an element to . this.items
- The `pop()` method removes the last element from . this.items
- The `length` property gives the length of . this.items

JavaScript Program to Implement a Queue

If you are not familiar with **Programming** you maybe think about the queue in shop or hospital. But if you are a **programmer** you associate it 99 with **Data Structures** and **Algorithms**. Whoever you are, today we will discuss how to implement **Queue Data Structure in JavaScript** and what a its **differences with a simple Array**.

A queue is a data structure that follows **First In First Out (FIFO)** principle. The element that is added first is accessed at first. This is like being in queue to get a movie ticket. The first one gets the ticket first.

Implementation

Key Methods

1. enqueue(value)
2. dequeue()
3. print()

Auxiliary Methods

1. isEmpty()
2. getHead()
3. getLength()

Implementation

Before we start to write the code let's discuss the main principle of the **Queue Algorithm**. It works on the principle of **FIFO**. It means **First In First Out**. is just like a real queue of people in a supermarket.

If we continue our comparison, people in the queue are **Nodes**. And primarily we need to create the sample of the **Node**. We will use classes that are th main part of **OOP (Object-oriented programming)**.

Let's see how the **class Node** looks.

```
class Node {
    constructor(value) {
```

```
        this.value = value;
        this.next = null;
    }
}
```

This class consists of two parameters.

1. **this.value** — the value which Node stores
2. **this.next** — the link to the next Node in Queue (initially null, since there are no nodes in Queue)

Okay, we have already created the Node. But we also need to create a class which will store these Nodes and perform some actions on them.

```
class Queue {
    constructor() {
        this.head = null;
        this.tail = null;
        this.length = 0;
    }
}
```

Class Queue has three parameters.

1. **this.head** — the link to the first node in Queue
2. **this.tail** — the link to the last node in Queue
3. **this.length** — the number of nodes in Queue

The `Queue` class is created to implement the queue data structure. The class includes methoc like `enqueue()` , `dequeue()` , `peek()` , `isEmpty()` , `size()` , and `clear()` .

A `Queue` object is created using a `new` operator and various methods are accessed through the object.

- Initially, `this.items` is an empty array.
- The `push()` method adds an element to . this.items
- The `shift()` method removes the first element from . this.items
- The `length` property gives the length of . this.items

Key Methods

It's cool. Now we have everything we need to start writing **Queue Data Structure methods**.

The first method which we will consider is **enqueue(value)**.

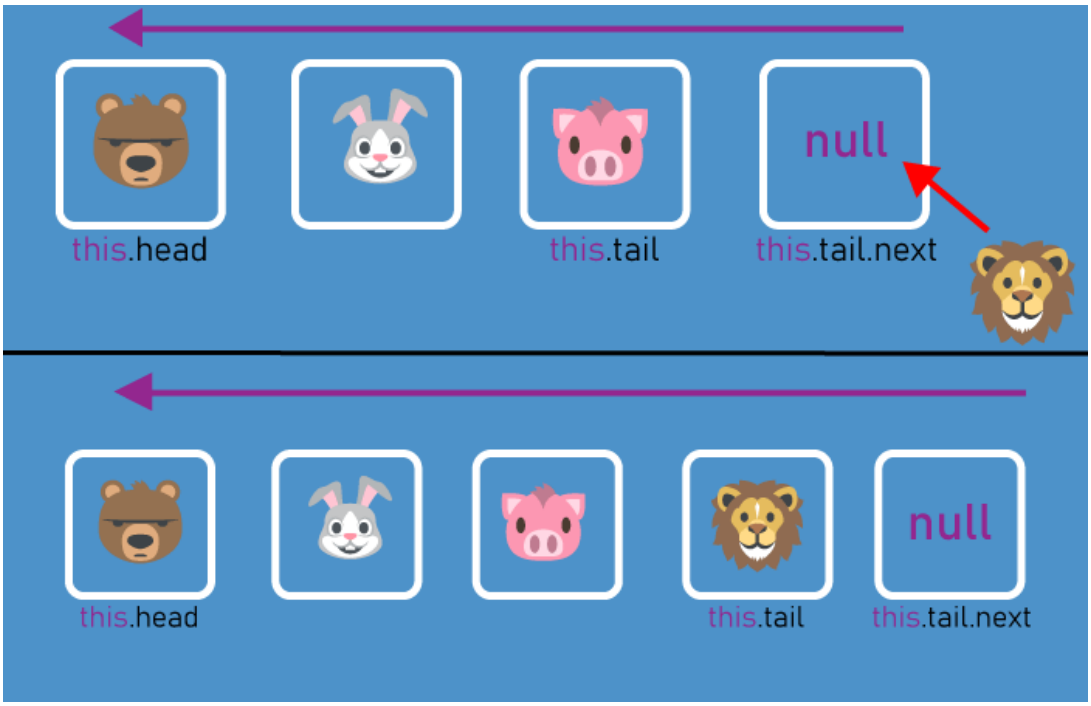
enqueue(value)

It needs in order to add the **Node** to the **tail (end)** of our **Queue**.

```
enqueue(value) {
    const node = new Node(value); // creates the node using class Node

if (this.head) { // if the first Node exitsts
   this.tail.next = node; // inserts the created node after the tail of our Queue
   this.tail = node; // now the created node is the last node
}  else { // if there are no nodes in the Queue
   this.head = node; // the created node is a head
  this.tail = node // also the created node is a tail in Queue because it is single.
    }

   this.length++; // increases the length of Queue by 1
```

```
}
```

The most difficult moment in the code above is statements in if {}. If you look at the picture below it will be easier to understand the meaning.



Okay, now we can add Nodes to the Queue. But it doesn't have to be endless (but sometimes in hospitals and supermarkets it is so). Let's learn ho to remove Nodes from the Queue.

dequeue()

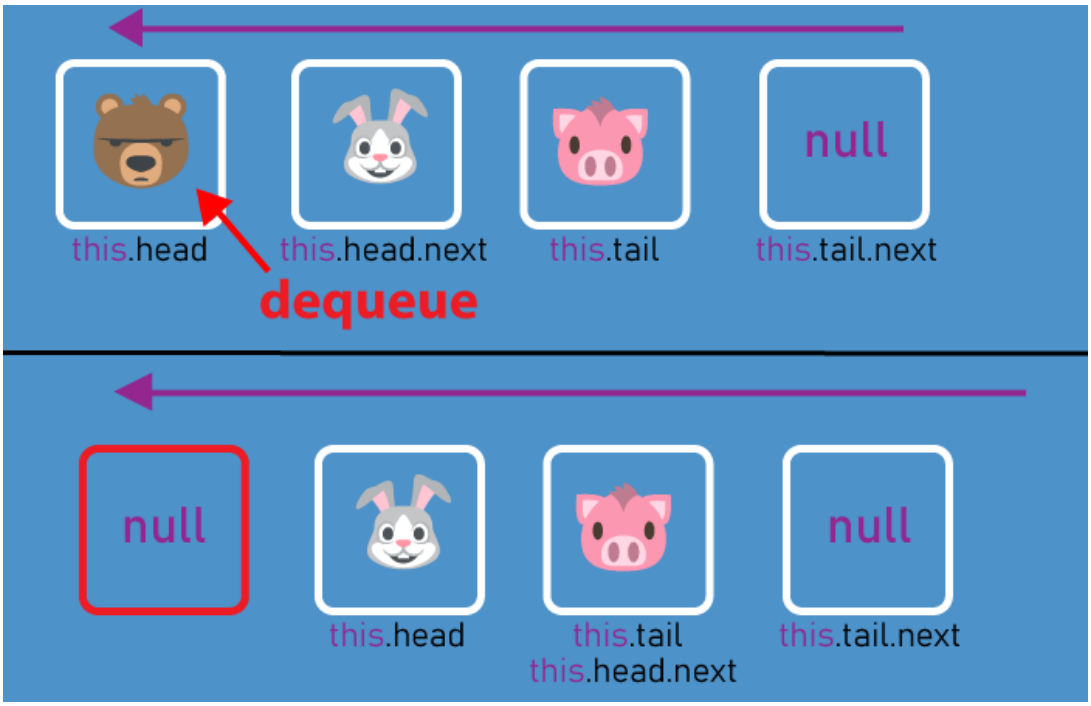
```
dequeue() {
  const current = this.head; // saves the link to the head which we need to remove
  this.head = this.head.next; // moves the head link to the second Node in the Queue
  this.length--; // decreaments the length of our Queue

  return current.value; // returns the removed Node's value
}
```

It may be hard to understand the following string of code:

```
this.head = this.head.next;
```

- Let's remember the example from the real world. If the cashier punched the product, the satisfied customer leaves the queue and then the next customer goes.
- In our code, **this.head** is the satisfied customer who has already bought products.
- **this.head.next** is the next customer who becomes the head of the queue after the satisfied customer leaving.

Let's look at the picture for a complete understanding.

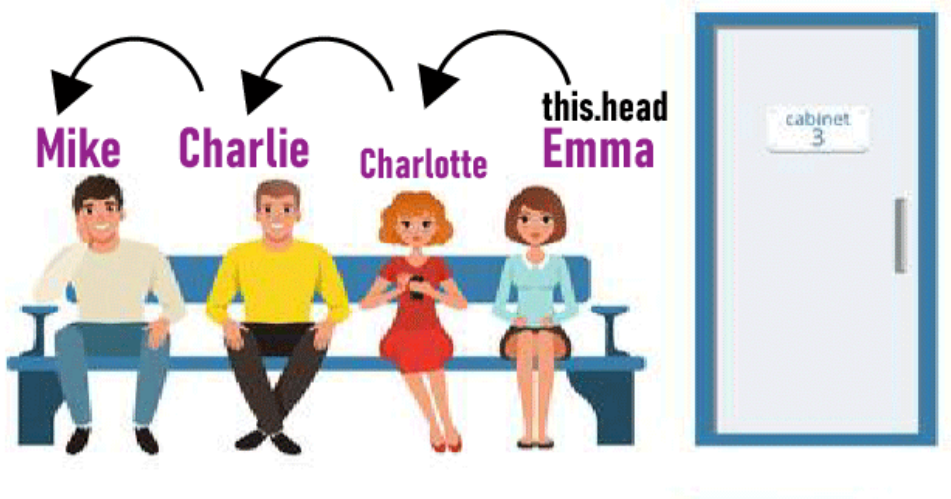


So now we can add and remove nodes from the queue. But we only know information about the first and last nodes (or people, if compared to the re world). We certainly want to know what happens in the middle of the **Queue**. To do this, let's create **print()** method which will **show all the values** all **Nodes** in the **Queue**.

print()

```
print() {
    let current = this.head; // saves a link to the head of the queue

    while(current) { // goes through each Node of the Queue
        console.log(current.value); // prints the value of the Node in console
        current = current.next; // moves link to the next node after head
    }
}
```

In order to understand it, let's imagine that the person is this.head (current) and his name is current.value. Okay, the first man in the queue asks th name of the second. Then the second person asks the name of the third person and the same until the end of the Queue. The print()method works th same way.




```
console.log('Emma');
console.log('Charlotte');
console.log('Charlie');
console.log('Mike');
```

Auxiliary Methods

In order to extract additional information from the **Queue**, we will create 3 auxiliary methods.

The first is **isEmpty()**.

isEmpty()

```
isEmpty() {
    return this.length === 0;
}
```

This method simply checks whether there are **Nodes** in our queue or not. It returns **true** if there is at least one **Node** in **Queue** and **false** if there a no **Nodes**.

getHead()

```
getHead() {
    return this.head.value;
}
```

This method returns the value of the first **Node** in the **Queue**.

getLength()

```
getLength() {
    return this.length;
```

```
}
```

It returns the number of **Nodes** in our **Queue**.

Why do you need a Queue? What are the differences with Array?

Indeed, why do we need to write the code for the **Queue** if we can use **JavaScript Arrays**? The answer is **Time Complexity**. Let's compare the **big O** the **Queue** and **Arrays**. Look at the table below.

	Access	Search	Insertion (at the end)	Deletion (from the beginning)
Queue	O(n)	O(n)	O(1)	O(1)
Array	O(1)	O(n)	O(1)	O(n)

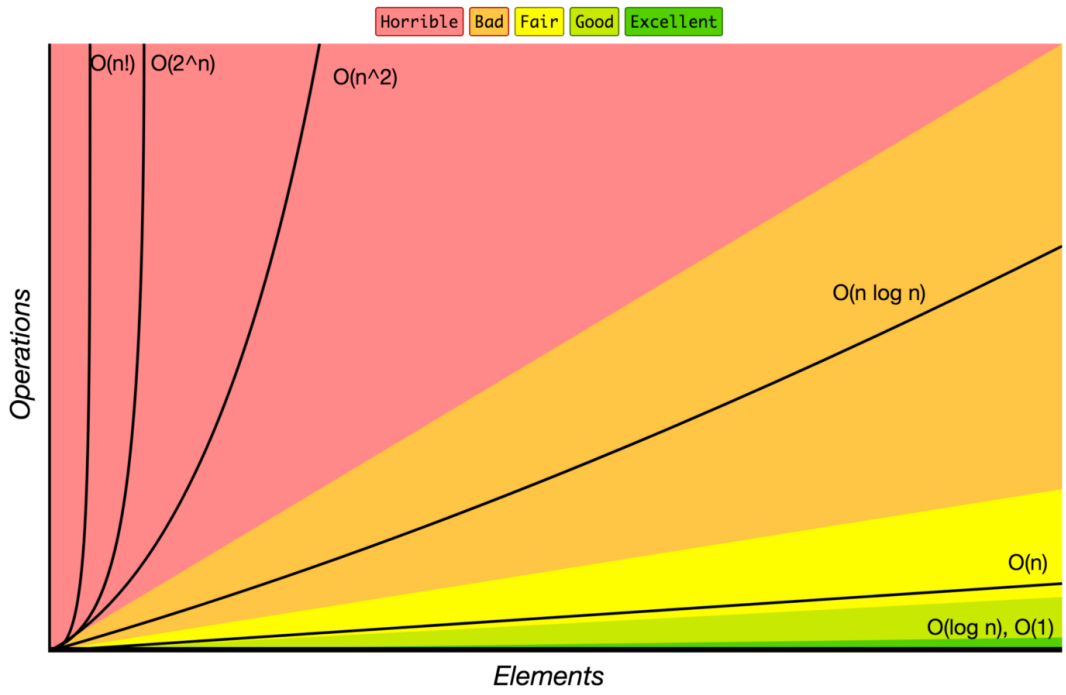
As you can see if we want to remove an element from the beginning of the array we need to do ***n* operations** where ***n*** is **the number of elements** in th array. While **Queue** needs only **1 operation** to do the same.

The problem with an array is that it has to move each element, **decrementing each index by 1**. In the **Queue Algorithm**, we simply move the link of th head.

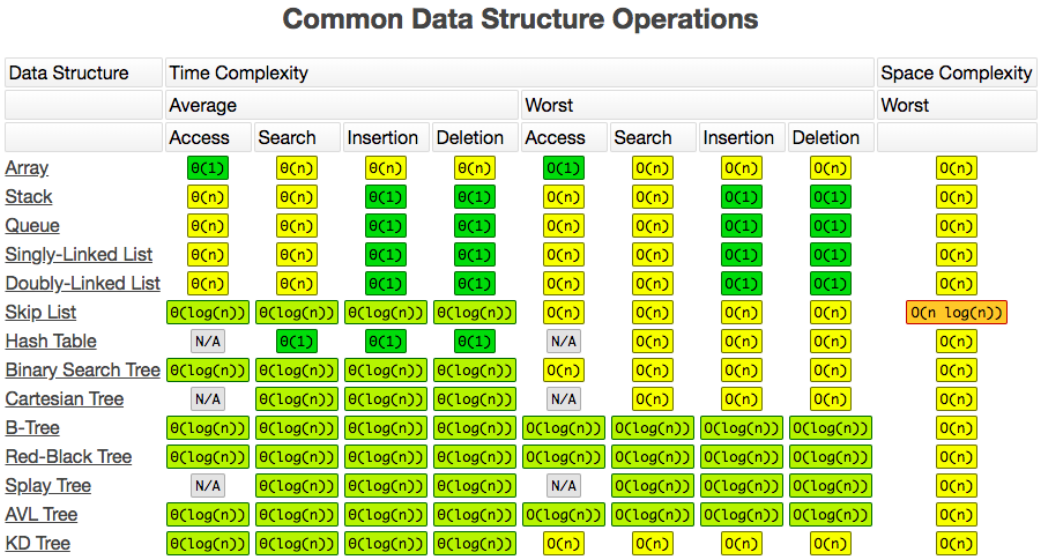
Big O of Queues

Operation	Time Complexity
Insertion	0(1) / Constant Time
Removal	0(1) / Constant Time
Searching	0(N) / Linear Time
Access	0(N) / Linear Time

Big-O Complexity Chart



Common Data Structure Operations



Queue Complexity Analysis

Queue from Array

Operation	Time Complexity
Enqueue	0(N) / Linear Time
Dequeue	0(1) / Constant Time

Queue from Linked List

Operation	Time Complexity
Enqueue	0(1) / Constant Time
Dequeue	0(1) / Constant Time

Thank you !