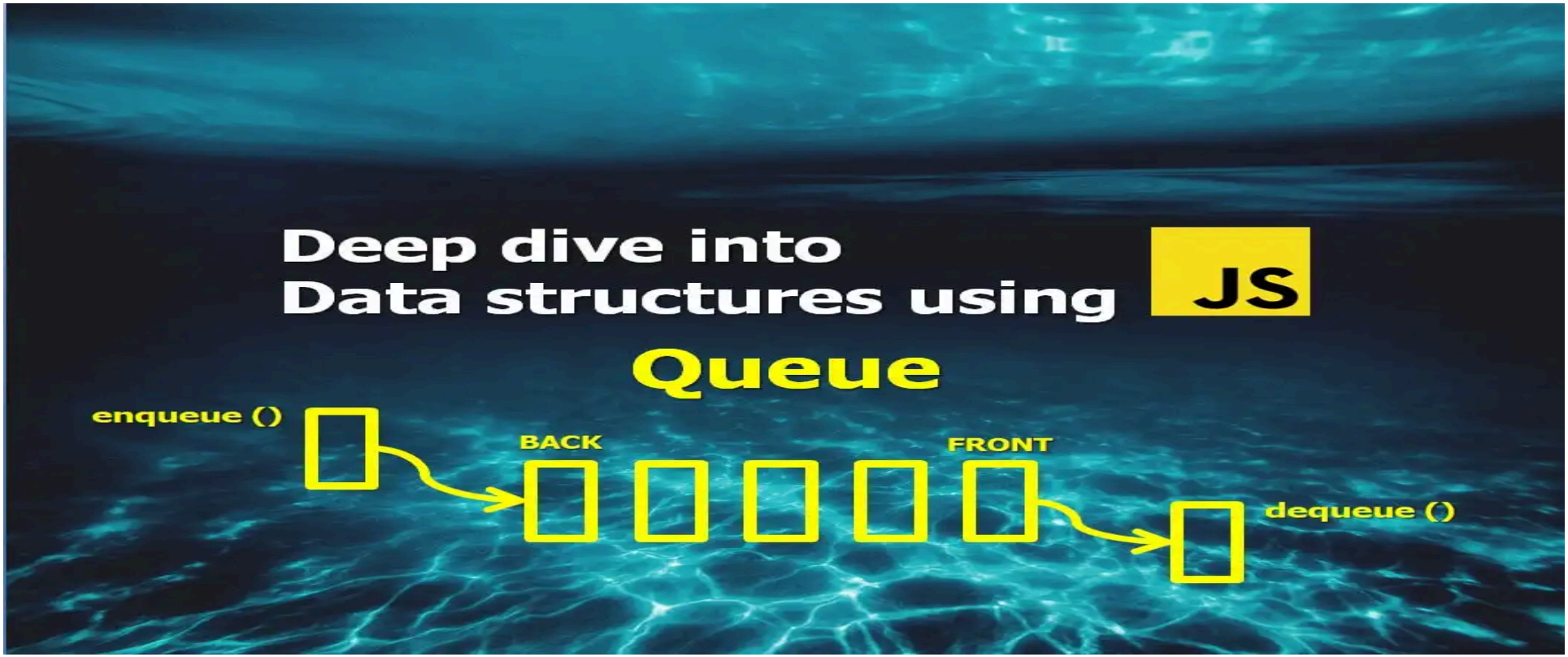
Deep Dive into Data structures using Javascript - Queue



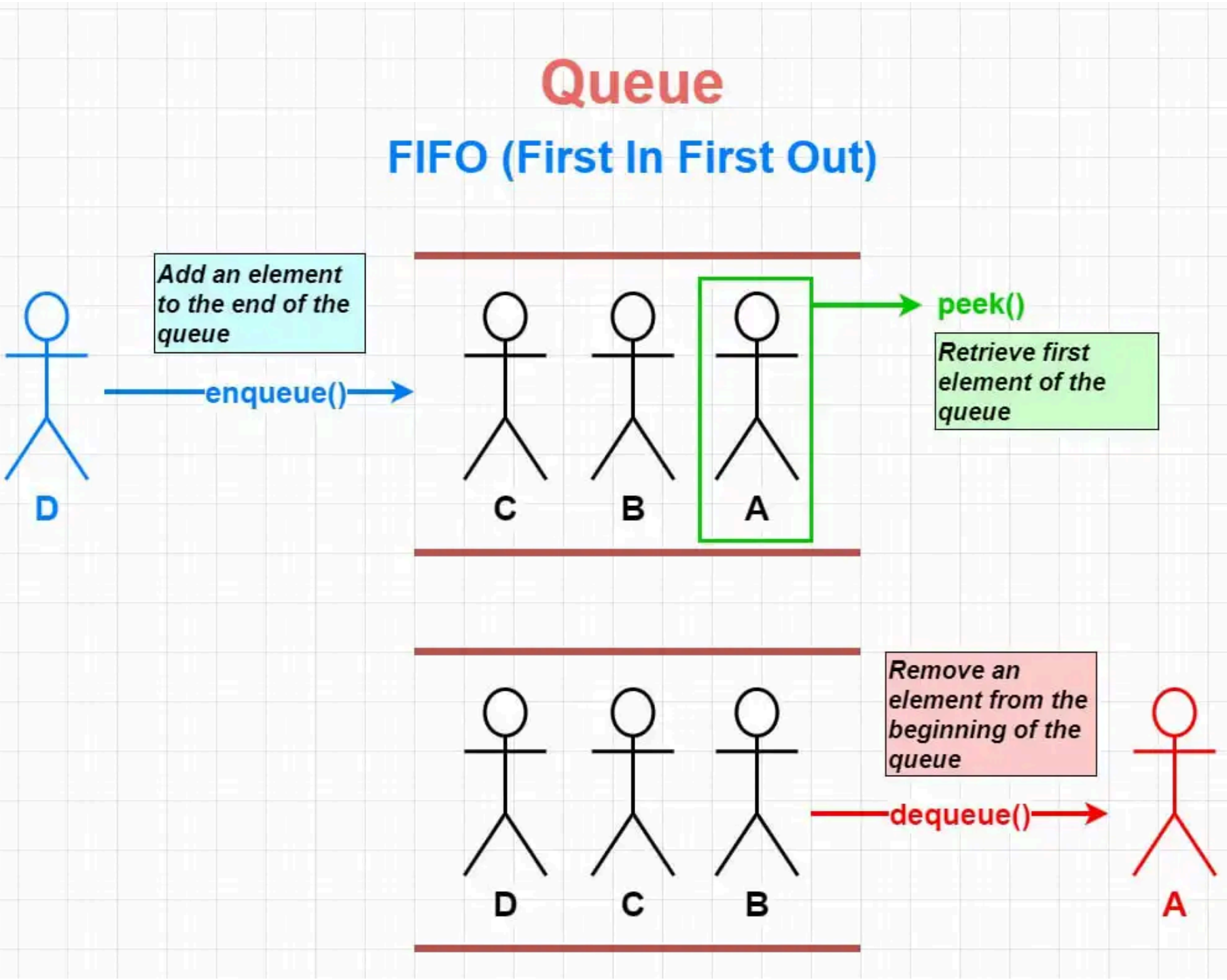
What is a Queue?

Queue is a linear data structure that stores it's elements in sequential order similar to Arrays. When adding or removing elements, it follows a particular order called **FIFO** - which is short for **First In First Out**.

An easy way to understand / memorize how Queue operates with **FIFO** is imagining a literal queue, in other words a group / queue of persons in a waiting line. Each new person starts at the end of the list (enqueue), and the first person that came into the waiting line is the first one that goes out from the line. All persons follows the same order until there is no one left.

Queue is not a built-in data structure in Javascript, but it is quite simple to implement a custom one.

Anatomy of a Queue


Queue
FIFO (First In First Out)
Add an element to the end of the queue
enqueue()
D
C
B
A
peek()
Retrieve first element of the queue
Remove an element from the beginning of the queue
dequeue()
D
C
B
A

Queue data structure has quite basic and straight-forward anatomy and it is very similar to the Stack data structure - we store and manage the elements in sequential order using a few methods. Whenever we want to add (enqueue) an item, we add to the end of the list like Stacks, but when we do remove (dequeue) an item, we always work at the very beginning of the list to follow the FIFO (First In First Out) order of operations. Similarly, with the peek method we can quickly retrieve the first inserted element in the Queue. Key difference between Queue and Stack is the place where do we do the removals in a list: with Queue we always remove from the beginning, with Stack the removal is always at the very end.

When to use Queue

Let's start with taking a quick look at the Big O of common operations in Queue:

Method	Worst case
enqueue()	O(1)
dequeue()	O(1)
peek()	O(1)
lookup (access / search) - optional	O(n)

Queue is useful when you specifically want to enforce FIFO (First In First Out) rule of access. A good example would be any type of waiting line where you want to operate in "first come first serve" order. It could be anything from selling products that has a limited stock (so the first

customer that is in line that gets the product in case there is only one left) to booking appointments or hotel rooms.

Similar to Stack, Queue also has very few methods (enqueue, dequeue and peek). This enables enforcing constrained access to data using the FIFO (First In First Out) rule, and few methods to deal with will also make the code easier to understand and work with.

You can also have an optional lookup / iterator method - but if you need to use it very often, you might reconsider using an Array instead. Because the primary usage of a Queue is not frequent iterations, so it won't suit well from a conceptual perspective.

Queues are used in many programming languages and applications in real world. Some examples are:

- Used in **CPU Task Scheduling** - which is a mechanism to determine the process that will use the CPU resources while another process is being on hold.

- Used in printers - each print request goes into a queue and the first request in line being executed while other requests are being on hold.

- Also used in BFS (Breadth First Search / Traversal) which has usage in advanced data structures like Trees and Graphs, Level Order Search in Binary Trees, problem solving concepts like finding shortest path in a maze / matrix and detecting palindromes.

Queue implementation in Javascript

Queues can be implemented using an Array or a Linked List. Linked List is a better choice due to better performance and proper time complexity match on removals (dequeue). Because we do the removals from the beginning of the list - this action costs Constant time O(1) in Linked List (time complexity for dequeue is supposed to be O(1)) while it costs Linear time O(n) in Arrays due to index shifting. Whenever an element is being removed from the Array (as long as it is not the last element in the list), all the following elements are shifted to match the correct index number due to size change in the list.

So you can also choose to go with the Array version depending

On the other hand, Array version is simpler to implement compared to the Linked List version, it also works quite well when you deal with small set of items. So you can also choose to go with the Array version depending on the use case - as long as you are sure that you won't need to scale the amount of items that you are dealing with. If you are not sure, go with the Linked List version - because the performance difference will be huge as soon as scaling comes into the picture.

Therefore we maintain both head and tail pointers.

I will be sharing both implementations below. For both of them we will be using an ES6 Class to build this data structure. Here is the list of methods in the implementation:

- `enqueue(value)` - *add to the end of Queue.*
- `dequeue()` - *remove from the beginning of Queue*
- `peek()` - *retrive first element in the Queue*
- `isEmpty()` - *helper method to check if Queue is empty*
- `length()` - *optional method to check the length of Queue*
- `clear()` - *optional method to clear the Queue*
- `toArray()` - *optional method returns Queue elements as an Array*

I hope this article helped you to understand what Queue is and how it works! I'd like to encourage you to experiment with the implementation below in your favorite code editor. Thanks for reading!

Queue implementation using Linked List:

```
// Minimal Linked List implementation with: append, deleteHead and toArray (optional) methods.
// To match the time complexity of O(1) on both add (enqueue) & removal (dequeue) of Queue, we will be using both ends of the Linked List. T
```

Therefore we maintain both head and tail pointers.

```
class LinkedListMinimal {
  constructor(value) {
    this.head = null
    this.tail = null
    this.length = 0
  }

  // add to the end of the list
  append(value) {
    // Initialize a newNode with value recieved and next as null.
    const newNode = {
      value: value,
      next: null
    }

    // Let's check if Linked List is empty or not first.
    if (!this.head) {
      // If there is no head (no elements) it is empty. In that case make the newNode as head
      // since it is the only node at this point and there is no tail either,
      // tail will also have the same value (both head and tail will point to same place in memory from now on)
      this.head = newNode
      this.tail = newNode
    } else {
      // If Linked List is not empty, Attach new node to the end of linked list:
      this.tail.next = newNode
      this.tail = newNode
    }

    this.length++
```

```
}

deleteHead() {
  if (!this.head) return

  const headVal = this.head.value

  // if one element left
  if (this.length === 1) {
    this.head = null
    this.tail = null
    this.length--
    return headVal
  }

  // define newHead as this.head.next
  const newHead = this.head.next
  // now change the head pointer to newHead
  this.head = newHead
  this.length--
  return headVal
}

// toArray - loop through nested objects, then return the values in an array
toArray() {
  const array = []
  // Initialize a currentNode variable pointing to this.head - which will be the starting point for traversal.
  let currentNode = this.head
```

```
    // fill the array until we reach the end of list:
    while (currentNode !== null) {
      array.push(currentNode.value)
      currentNode = currentNode.next
    }
    return array
  }
}

class Queue {
  constructor() {
    this.items = new LinkedListMinimal()
  }

  // add to the end of Queue
  enqueue(value) {
    this.items.append(value)
  }

  // remove from the beginning of Queue
  dequeue() {
    if (this.isEmpty()) {
      return null
    }

    const dequeuedItem = this.items.deleteHead()
    if (!dequeuedItem) return
    return dequeuedItem
  }
```

```
  // retrive first element in the Queue
  peek() {
    if (this.isEmpty()) return null
    return this.items.head.value
  }

  // helper method to check if Queue is empty
  isEmpty() {
    return this.items.length === 0
  }

  // optional method to check the length of Queue
  length() {
    return this.items.length
  }

  // optional method to clear the Queue
  clear() {
    this.items = new LinkedListMinimal()
  }

  // optional method returns Queue elements as an Array
  toArray() {
    return this.items.toArray()
  }
}
```

Queue implementation using Array:

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

  // add to the end of Queue
  enqueue(item) {
    this.items.push(item)
  }

  // remove from the beginning of Queue
  dequeue() {
    if (this.isEmpty()) return
    // This costs Linear Time O(n) instead of O(1) due to array shift:
    const dequeuedItem = this.items.shift()
    return dequeuedItem
  }

  // retrive first element in the Queue
  peek() {
    if (this.isEmpty()) return
    return this.items[0]
  }

  // helper method to check if Queue is empty
  isEmpty() {
    return this.items.length === 0
```

```
  }

  // optional method to check the length of Queue
  length() {
    return this.items.length
  }

  // optional method to clear the Queue
  clear() {
    this.items = []
  }

  // optional method returns Queue elements as an Array
  toArray() {
    return this.items
  }
}
```