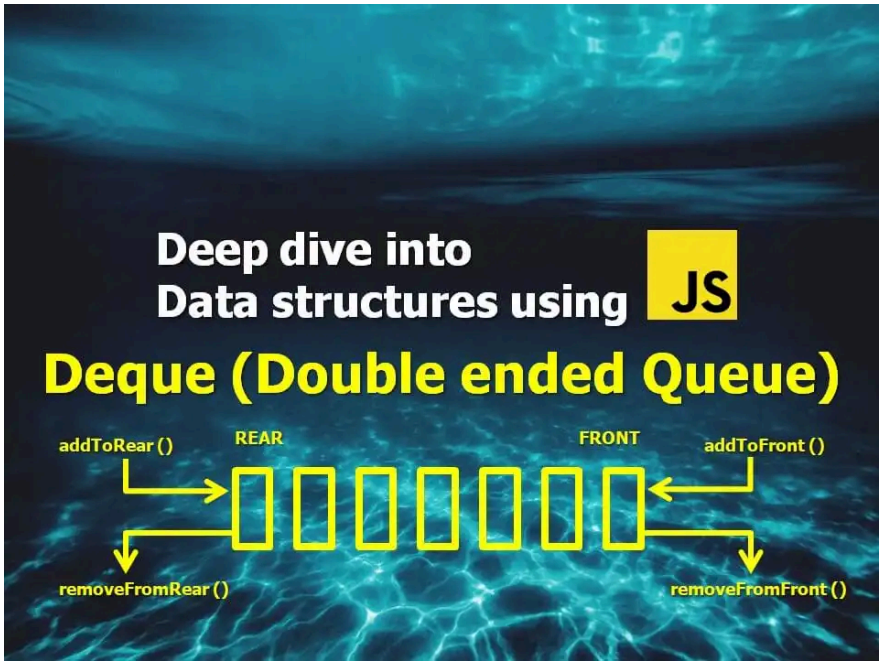
Deep Dive into Data structures using Javascript - Deque (Double ended Queue)



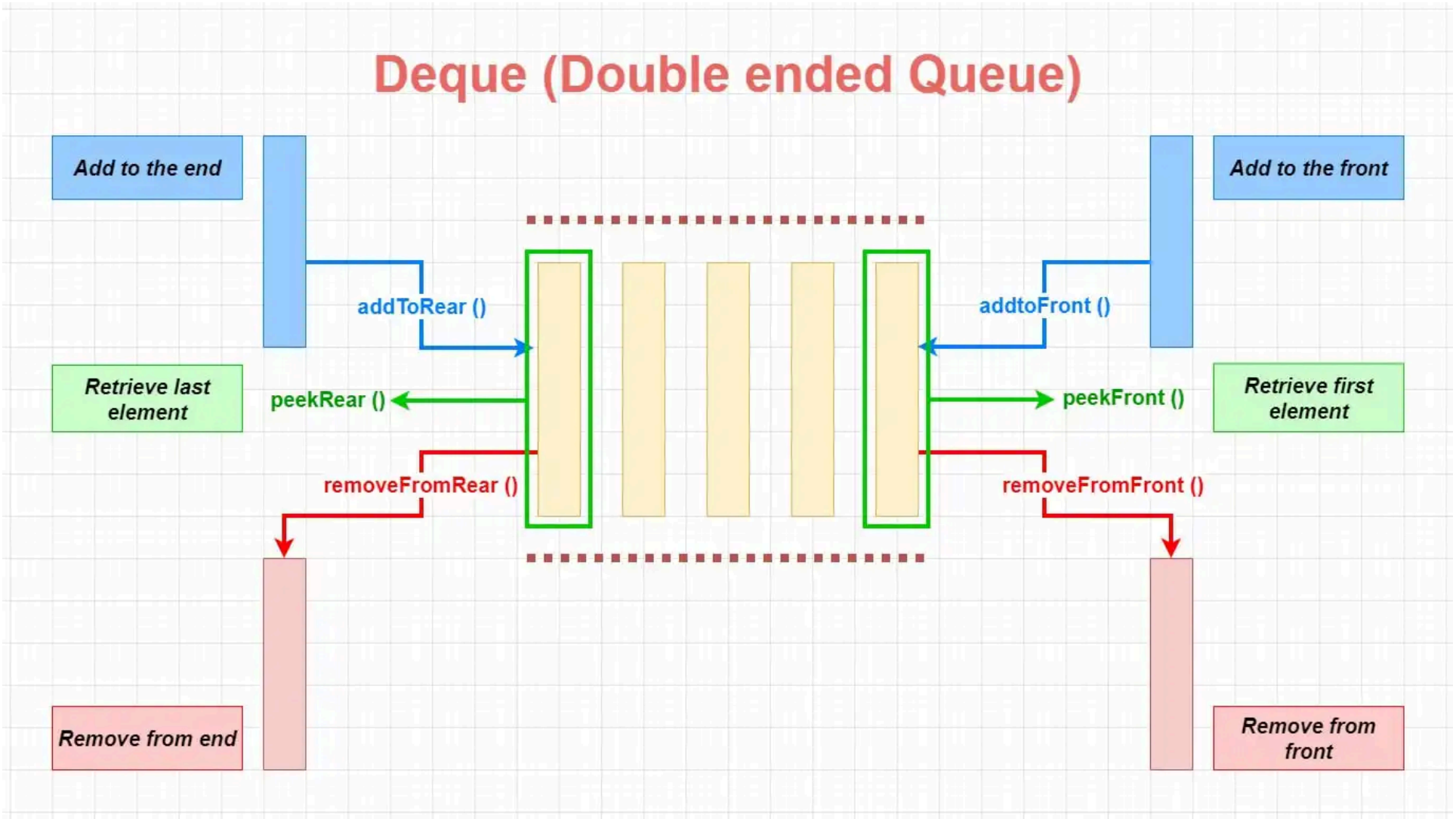
What is a Deque?

Deque, which can be also called as Double ended Queue is a variation of Queue data structure with extended abilities. It is in fact a combination of Queue and Stack data structures - Deque supports FIFO (First In First Out) type of operations like a Queue and LIFO (Last In Last Out) type of operations like a Stack without enforcing the rules. Because of this combination, Deque can also be used as a Queue or a Stack alternative.

Similar to Queues and Stacks, Deque operations avoids working at the middle of a list. Deque is known for it's efficiency and speed on performing additions or removals at the either end of a list.

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

Anatomy of a Deque



Deque is a linear data structure that is fairly easy to understand. We store and manage the elements in sequential order and only work at the beginning or at the end of a list as we see above. The rule of enforcement is another thing that differentiates Deque from a Queue or Stack. Despite the fact that you can use Deque as either one of them, Deque does not force an order of operation like FIFO (First In First Out) or LIFO (Last In First Out) out of the box. So if you want to use Deque as a Queue or Stack, you will need to customize the restrictions accordingly - for example you can only use the methods that follows a specific order.

When to use Deque

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

Method	Worst case
addToFront()	O(1)
addToRear()	O(1)
removeFromFront()	O(1)
removeFromRear()	O(1)
peekFront()	O(1)
peekRear()	O(1)
lookup (access / search) - optional	O(n)

Since Deque is a combination of Queue and Stack, you can use the Deque for both cases where they are useful. For more details, please refer to the sections *"When to use Queue"* in the article: Queue and *"When to use Stack"* in the article: Stack

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

Apart from Queue and Stack use cases, Deques are also used in many programming languages and applications in real world. Some examples are:

- A waiting list that works with both ends. If we imagine a classic waiting line, we know that it doesn't operate with only "first come first serve" order. There is always a possibility some people will leave from the end of the waiting line before their turn comes (specially if they see a disheartening long line). With Deque we can remove elements from both ends unlike Queue where we are only able to remove from the beginning.
- Applications or problems that has use of clockwise and anticlockwise rotations, for example: rotating the positions of a matrix problem.
- Used in *Sliding window* problem solving technique, which usually applies to Array or List type of problems. Deque helps to reduce time complexity from O(n^2) Quadratic time to O(n) Linear time by replacing brute force nested loop solutions with a single loop.
- Used in *A-Steal job scheduling algorithm*, which is used for implementing task scheduling in a multiprocessor environment.

Deque implementation in Javascript

Deques can be implemented using an Array or a Doubly Linked List. Doubly Linked List is a better choice due to better performance and proper time complexity match on additions and removals at the beginning of the list. Because when we do the additions and removals from the beginning of a list - this action costs Constant time O(1) in a Doubly Linked List (time complexity for addToFront() and removeFromFront() 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.

Now you maybe thinking *"Why do we need to use Doubly Linked List instead of a regular Linked List?"* - the answer is because Doubly Linked list has better performance on removals from the end of a list compared to regular one. Removing tail costs Linear time O(n) in a regular Linked List while it costs Constant time O(1) in a Doubly Linked List.

On the other hand, Array version is simpler to implement compared to the Doubly 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 Doubly Linked List version - because the performance difference will be huge as soon as scaling comes into the picture.

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:

- `addToFront(value)` - *add to the beginning of Deque.*
- `addToRear(value)` - *add to the end of Deque.*
- `removeFromFront()` - *remove from the beginning of Deque*
- `removeFromRear()` - *remove from the end of Deque*
- `peekFront()` - *retrive first element in the Deque*
- `peekRear()` - *retrive last element in the Deque*
- `isEmpty()` - *helper method to check if Deque is empty*
- `length()` - *optional method to check the length of Deque*
- `clear()` - *optional method to clear the Deque*
- `toArray()` - *optional method returns Deque elements as an Array*

I hope this article helped you to understand what Deque 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!

Deque implementation using Doubly Linked List:

```
// Minimal Doubly Linked List implementation with: append, prepend, deleteHead, deleteTail and toArray (optional) methods.
// We will be only using the methods that helps to perform additions and removals from both ends.
class DoublyLinkedListMinimal {
  constructor(value) {
      this.head = null
      this.tail = null
      this.length = 0

      // make it optional to create Doubly Linked List with or without starter value
      if (value) {
          this.append(value)
      }
  }

  // Add to the end of list
  append(value) {
      // Initialize a newNode with value recieved
    const newNode = {
      value,
      next: null,
      prev: null
    }

      // Let's first check if Doubly Linked List is empty or not.
      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 {
            // Since the newNode will be the new tail, set the prev value to current tail before applying changes. Timing is important!
            newNode.prev = this.tail
            // we have this.tail = this.head is setup with first entry
            // at first we populate the this.tail.next with newNode. Since both are referencing the same object, both head and tail will look equal at this step:
            this.tail.next = newNode
            // at this step, we cleanup the tail by setting it to newNode. In other words we extended the head by using tail first, then cleaned up the tail by using newNode.
            this.tail = newNode
        }
        this.length++
        return this
    }

    // Add to the beginning of list
    prepend(value) {
        // Let's check first if Doubly Linked List is empty or not.
        // If that's the case, return here by using the append method instead

        if (!this.head) {
            return this.append(value)
        }

        // Initialize a newNode with value recieved
        const newNode = {
          value,
          next: null,
          prev: null
        }
```

```
    // apply a reference to newNode.next prop. When we add it at the start, naturally prepended node's
    newNode.next = this.head
    // Since the newNode will be the new previous for the current head, set the prev value of head to t
    this.head.prev = newNode
    // now that newNode has the this.head as next and newNode as prev, we can set the this.head as newN
    this.head = newNode
    this.length++
    return this
}

// toArray - loop through nested objects, then return the values in an array
toArray() {
    const array = []
    let currentNode = this.head

    while (currentNode !== null) {
        array.push(currentNode.value)
        currentNode = currentNode.next
    }
    return array
}

// Delete from beginning of list
deleteHead() {
    // check the length - if zero return a warning
    if (this.length === 0) return 'List is empty'

    // If there is only one node left:
    if (this.length === 1) {
```

next value should point to the this.head.

be newNode. We do this before changing the pointer of this.head to newNode. Timing is important!

Node directly.

```
            const headVal = this.head.value
            this.head = null
            this.tail = null
            this.prev = null
            this.length--
            return headVal
        }

        // pick the current head value:
        const headVal = this.head.value
        // define newHead as this.head.next
        const newHead = this.head.next
        // make the new heads prev pointer null
        newHead.prev = null
        // now change the head pointer to newHead
        this.head = newHead
        this.length--
        return headVal
    }

    // Delete from the end of list
    deleteTail() {
        // check the length - if zero return a warning
        if (this.length === 0) return 'List is empty'

        // If there is only one node left:
        if (this.length === 1) {
            const tailVal = this.tail.value
            this.head = null
```

```
            this.tail = null
            this.prev = null
            this.length--
            return tailVal
        }

        // Define new tail by traversing to previous Node of tail idx
        // Note that, tail always points to null. (which is length).
        // length - 1 will point to last Node with a value. Therefore we need to target length - 2
        const tailVal = this.tail.value
        const newTail = this.tail.prev
        // Now, we can just simply update the pointer of newTail to null:
        newTail.next = null
        this.tail = newTail
        this.length--
        return tailVal
    }
}

class Deque {
    constructor() {
        this.items = new DoublyLinkedListMinimal()
    }

    addToFront(item) {
        this.items.prepend(item)
    }

    addToRear(item) {
```

```
  this.items.append(item)
}

removeFromFront() {
  const removedItem = this.items.deleteHead()
  return removedItem
}

removeFromRear() {
  const removedItem = this.items.deleteTail()
  return removedItem
}

peekFront() {
  return this.items.head.value
}

peekRear() {
  return this.items.tail.value
}

isEmpty() {
  return this.items.length === 0
}

length() {
  return this.items.length
}
```

```
  clear() {
    this.items = new DoublyLinkedListMinimal()
  }

  toArray() {
    return this.items.toArray()
  }
}
```

Deque implementation using Array:

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

  addToRear(item) {
    this.items.push(item)
  }

  addToFront(item) {
    this.items.unshift(item)
  }

  removeFromFront() {
    const removedItem = this.items.shift()
    return removedItem
```

```
}

removeFromRear() {
  const removedItem = this.items.pop()
  return removedItem
}

peekFront() {
  return this.items[0]
}

peekRear() {
  return this.items[this.items.length - 1]
}

isEmpty() {
  return this.items.length === 0
}

length() {
  return this.items.length
}

clear() {
  this.items = []
}

toArray() {
  return this.items
```

```
    }
}
```