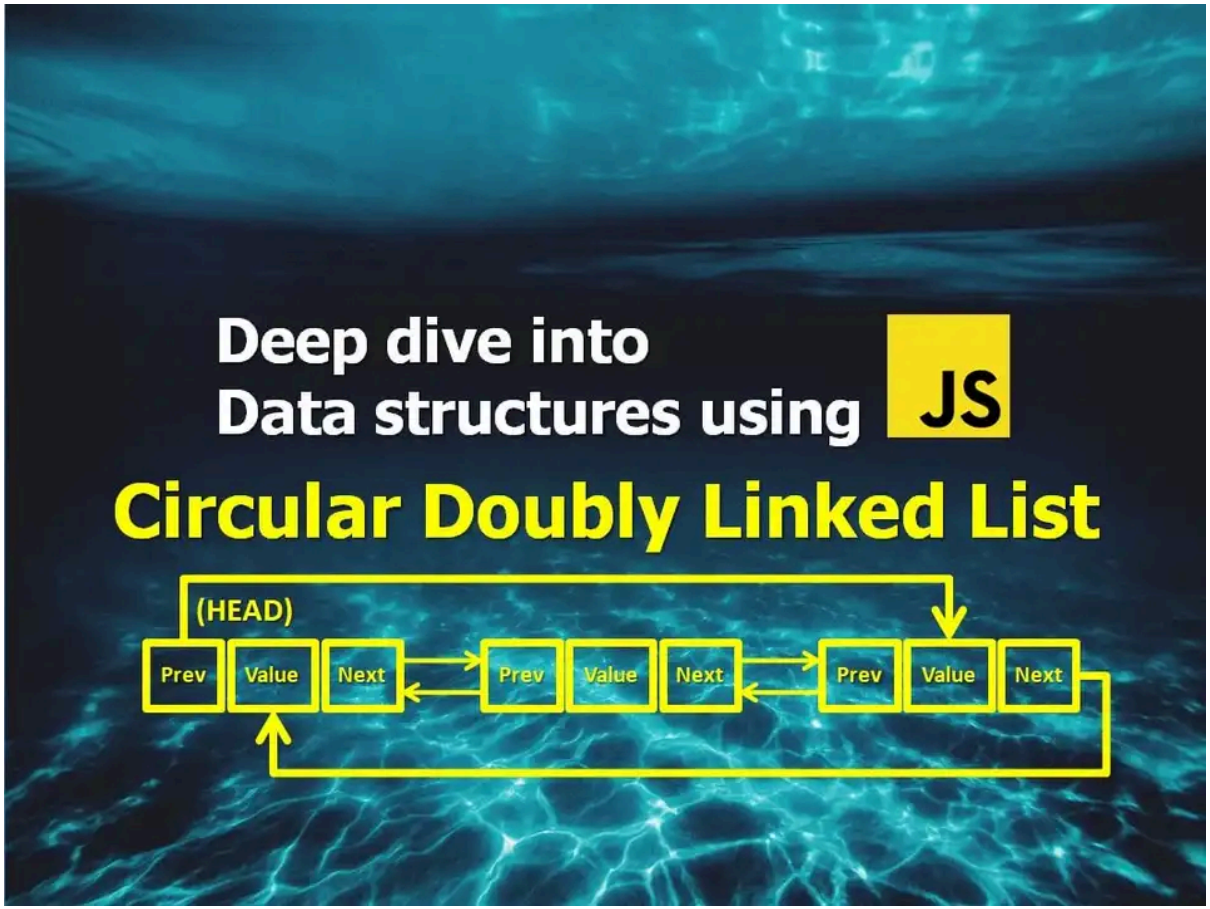
Deep Dive into Data structures using Javascript - Circular Doubly Linked List



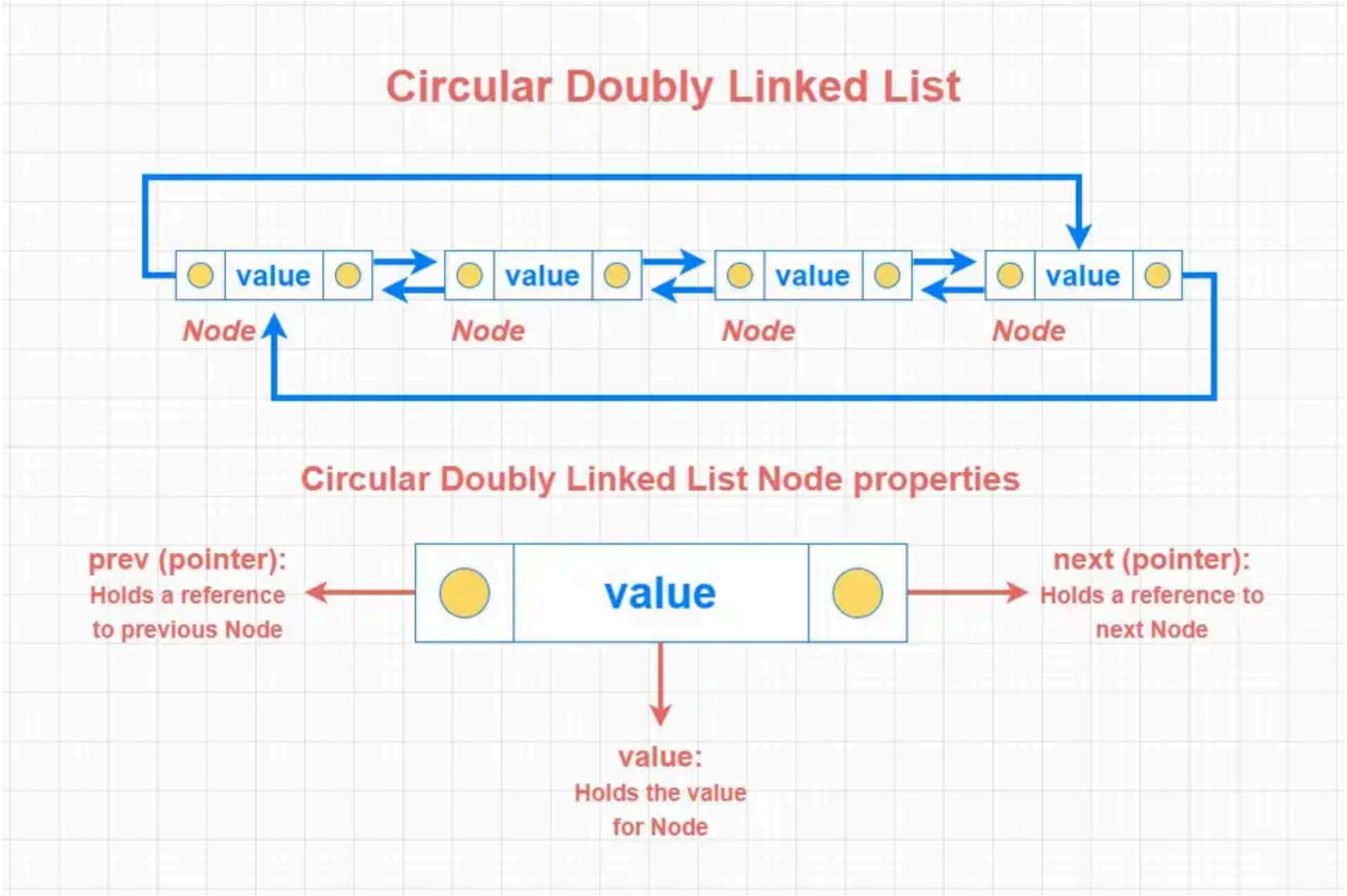
What is a Circular Doubly Linked List?

A Circular Doubly Linked list is a variation of Linked List data structure, which is very similar to Singly Circular Linked List with only one difference: nodes contains 2 pointers (previous and next) instead of a single one (next). We can also say that the difference between them follows the similar fashion like Linear Singly Linked List vs Linear Doubly Linked List.

In Circular Doubly Linked List, the tail and head points to eachother to form a circle of nodes - we do not have any null termination at the tail. Because of this structural difference, Circular Doubly Linked list classifies as a non-linear data structure unlike the regular Linked list variations which is linear - since the shape of connected nodes forms a cyclic data structure.

The tone of this article will be assuming you are familiar with Linked List, Doubly Linked List and Circular Linked List data structures. If that's not the case or you need a quick refreshment on them, I would suggest you to first follow through the article links below, then come back and continue here later:

Anatomy of a Circular Doubly Linked List



A Circular Doubly Linked List is consisted by series of connected nodes that forms a cyclic data structure. Each node contains 3 properties:

Prev: Holds a reference (pointer) to the previous Node.

Value: Holds the value / data for the Node.

Next (pointer): Holds a reference (pointer) to the next Node.

Since this variation has no null termination and not a linear list, we don't actually have a "head" or the "tail" here with the same meaning. But still, maintaining pointers for first and last inserted elements will help to form the circular structure by linking them to eachother. Knowing first and last elements also makes it easier to implement the common methods.

Common methods of Circular Doubly Linked List is extremely similar with regular Doubly Linked List methods, but there are 2 cavaets that we need to be aware of:

- We cannot use the same traversal exit condition (null termination). If we are not careful, a traversal can end up in an infinite loop.

- Relationship of head and tail is different here, therefore we need to keep that in mind and make sure to maintain the circular reference of pointers properly. Specially whenever we do inserts or deletions, wrong way of updating pointers can cause losing rest of the list in memory.

When to use Circular Doubly Linked List

Circular Doubly Linked list shares the same Big O complexity with Doubly Linked Lists. And the reason we may want to consider choosing this data structure is exactly the same with the Singly Circular Linked List variant: Whenever we need to deal with operations that has circular logic - a list that accesses items over and over again in a loop. For more details, please refer to the section *"When to use Circular Linked List"* in the article: Circular Linked List

Now you might be thinking, *"If this circular data structure makes sense for my use case, how should I know which variant to choose between Circular Linked Lists - should I go with Singly or Doubly?"*

Answer to this would be pretty much same with deciding between Singly vs Doubly Linked Lists. Please refer to the section *"Doubly Linked List vs Singly Linked List"* in the article for more details: Doubly Linked List

Circular Doubly Linked List implementation in Javascript

We will be also using ES6 Classes to build this data structure similar to other Linked list types. Here is the list of methods in the implementation:

- `initialize(value)` - *helper for initializing the list with circular reference*
- `append(value)` - *add to the end*
- `prepend(value)` - *add to the beginning*
- `toArray()` - *return Circular Linked List elements in an array for ease of debugging*
- `traverseToIndex(index)` - *traversal helper*
- `insert(index, value)` - *add to the middle*
- `deleteHead()` - *delete from beginning*
- `deleteTail()` - *delete from the end*
- `delete(index)` - *delete from the middle*
- `reverse()` - *reverse order of items*

To make these methods easier to understand and reason with, I have placed code comments at specific places inside them. I hope this article helped you to understand how Circular Doubly Linked List works! I'd like to encourage you to experiment with the implementation below in your favorite code editor, and follow along with comments in case you need further understanding. Thanks for reading!

```
class Node {
    constructor(value) {
        this.value = value
        this.next = null
          this.prev = null
    }
}

class CircularDoublyLinkedList {
  constructor(value) {
    this.head = null
    this.tail = null
    this.length = 0

    if (value) {
      this.initialize(value)
    }
  }

  // used to initialize Circular Doubly Linked List
  initialize(value) {
    // create a node
    const newNode = new Node(value)
    // create a circular reference (points to itself)
    newNode.next = newNode
    newNode.prev = newNode
    // now make both head and tail to point on newNode
    this.head = newNode
    this.tail = newNode
```

```
    // increment length
    this.length++
  }

  append(value) {
    // if length is zero, use initialize method instead
    if (this.length === 0) {
      return this.initialize(value)
    }

    // create a new node
    const newNode = new Node(value)

    // fix newNode pointers to set it as a new tail:
    newNode.prev = this.tail
    newNode.next = this.head

    // now update the head and tail accordingly:
    this.head.prev = newNode
    this.tail.next = newNode

    // update the tail to be the newNode:
    this.tail = newNode
    // increment length:
    this.length++
  }

  prepend(value) {
    // if length is zero, use initialize method instead
```

```
    if (this.length === 0) {
      return this.initialize(value)
    }

    // create a new node
    const newNode = new Node(value)

    // fix newNode pointers to set it as a new head:
    newNode.next = this.head
    newNode.prev = this.tail

    // now update the head and tail accordingly:
    this.head.prev = newNode
    this.tail.next = newNode

    // update the head to be the newNode:
    this.head = newNode

    // increment length
    this.length++
  }

  // 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

    do {
```

```
      array.push(currentNode.value)
    currentNode = currentNode.next
    // NOTE:
    // Since there can be duplicate values in the list, we will be using "Referential equality" instead of comparing Node values as the exit condition (which is figuring out where the head is).
    // When strict equality operator is used in reference types in JS, it checks if compared values referencing the same object instance. This is useful when you like to compare references.
  } while (currentNode !== this.head)

  return array
}

// traverse to index
traverseToIndex(index) {
  if (index < 0) return undefined
  // keeps track of traversal
  let counter = 0
  // starting point
  let currentNode = this.head

  // traverse to the target index
  while (counter !== index) {
    currentNode = currentNode.next
    counter++
  }

  return currentNode
}

insert(index, value) {
  // if length is 0, just prepend (add to the beginning)
```

```
if (index === 0) {
  return this.prepend(value)
}
// validate the received index parameter:
if (!index) return 'Index is missing'
if (typeof index !== 'number') return 'Index should be a number'
if (index < 0) return 'Index should be bigger than zero'

// if length is too long, just append (add to the end)
if (index >= this.length) {
  return this.append(value)
}

// Initialize a newNode with value recieved.
const newNode = new Node(value)

// pick previous index
const preIdx = this.traverseToIndex(index - 1)

// pick target index
const targetIdx = preIdx.next

// Set the preIdx next to newNode. This is because newNode replaces the targetIdx's position.
preIdx.next = newNode

// Set the newNode prev to preIdx. This is because newNode replaces the targetIdx's position.
newNode.prev = preIdx

// Set the newNode next to targetIdx. This is because newNode replaces the targetIdx's position.
```

```
    newNode.next = targetIdx

    // Now, targetIdx (which have changed place until this step) will point the prev to the newNode. Again, timing is important on steps!
    targetIdx.prev = newNode
    this.length++
}

// remove from beginning:
deleteHead() {
    // check if there is a head value - if not return a warning or null
    if (this.length === 0) return null
    const currHead = this.head

    // if one element left
    if (this.length === 1) {
      const headVal = this.head.value
      this.head = null
      this.tail = 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

    // move the head to next node:
```

```
    this.head = newHead

    // set the tail next to new this.head:
    this.tail.next = this.head

    // set the new previous pointer to updated tail:
    this.head.prev = this.tail
    // decrement length:
    this.length--
    return headVal
}

// remove from end:
deleteTail() {
    // check the length - if zero return null
    if (this.length === 0) return null

    // 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
    }

    // store the tailVal (to return):
    const tailVal = this.tail.value
```

```
    // define new tail by picking the previous node of current tail.
    const newTail = this.tail.prev

    // prepare for replacement:
    // point newTail.next to this.head
    newTail.next = this.head
    // point this.head.prev to newTail. Now the new tail has correct pointers.
    this.head.prev = newTail

    // finalize removal by pointing current tail to newTail
    this.tail = newTail
    // decrement length:
    this.length--
    return tailVal
}

// Delete from specific index
    delete(index) {
        // if length is 0, just prepend (add to the beginning)
        if (index === 0) {
          return this.deleteHead()
        }

        // validate the received index parameter:
        if (!index) return 'Index is missing'
        if (typeof index !== 'number') return 'Index should be a number'

        // check the length - if zero return a warning
        if (this.length === 0) return 'List is empty'
```

```
// Validation - should not be less than 0
if (index < 0) return `Minimum idx should be 0 or greater`

// Check if it is the last element. In that case reset head and tail to null
if (this.length === 1) {
    const targetVal = this.head.value
    this.head = null
    this.tail = null
    this.prev = null
    return targetVal
}

// If not define removal style. Removal will be either head, middle or tail.
let removalType

if (index === 0) {
    removalType = 'head'
}
// When we do a removal from middle on Doubly Linked List, we need to take 3 indexes into account: pre, target and next.
// To be able to make it work the middle removal with the length prop,

 // we specify the comparison one minus form the length prop compared to a Singly Linked List.

 if (index >= this.length - 1) {
    removalType = 'tail'
}
if (index > 0 && index < this.length - 1) {
    removalType = 'middle'
```

```
    }

    if (removalType === 'head') {
        return this.deleteHead()
    }

    if (removalType === 'tail') {
        return this.deleteTail()
    }

    if (removalType === 'middle') {
        /*
       Pick the previous Node of targetIdx via traverse.
       Pick the target idx with preIdx.next
       Now make preIdx point to targetIdx next. This will remove the node in middle.
     */
        const preIdx = this.traverseToIndex(index - 1)
        const targetIdx = preIdx.next
        const targetVal = targetIdx.value
        const nextIdx = targetIdx.next
        preIdx.next = nextIdx
        nextIdx.prev = preIdx
        this.length--
        return targetVal
    }
}

reverse() {
  // Checkup - if list only contains one item, no need to reverse
```

```
if (this.length === 0) return
// do not reverse if there is a single element
if (this.length === 1) return

// We'll use 3 pointers. Prev and Next is empty at the start
let previousNode = null
let currentNode = this.head
let nextNode = null

// It is quite similar to doubly linked list reverse, the main difference is we can't use null termination as an exit condition - due to dealing with a circular list.

// To tackle this issue, we will be using "Referential equality" instead of comparing Node values as the exit condition (which is figuring out if we come back to the head node again).

// When strict equality operator is used in reference types in JS, it checks if compared values referencing the same object instance. This is useful when you need to compare references.
do {
     // Store next node reference
     nextNode = currentNode.next
       // Store prev node reference
     previousNode = currentNode.prev


       // Change next node of the current node so it would link to previous node.
     currentNode.next = previousNode
     currentNode.prev = nextNode

   // Move prevNode and currNode nodes one step forward.
     previousNode = currentNode
     currentNode = nextNode

   // console.log(previousNode.value, currentNode.value, nextNode.value)
} while (currentNode !== this.head)
```

```
        // Set the new tail with this.head (it contains the last item at this point of time):
        this.tail = this.head
              // Now reference this head to previousNode (contains the reversed list):
        this.head = previousNode

        return this.toArray()
    }
}
```