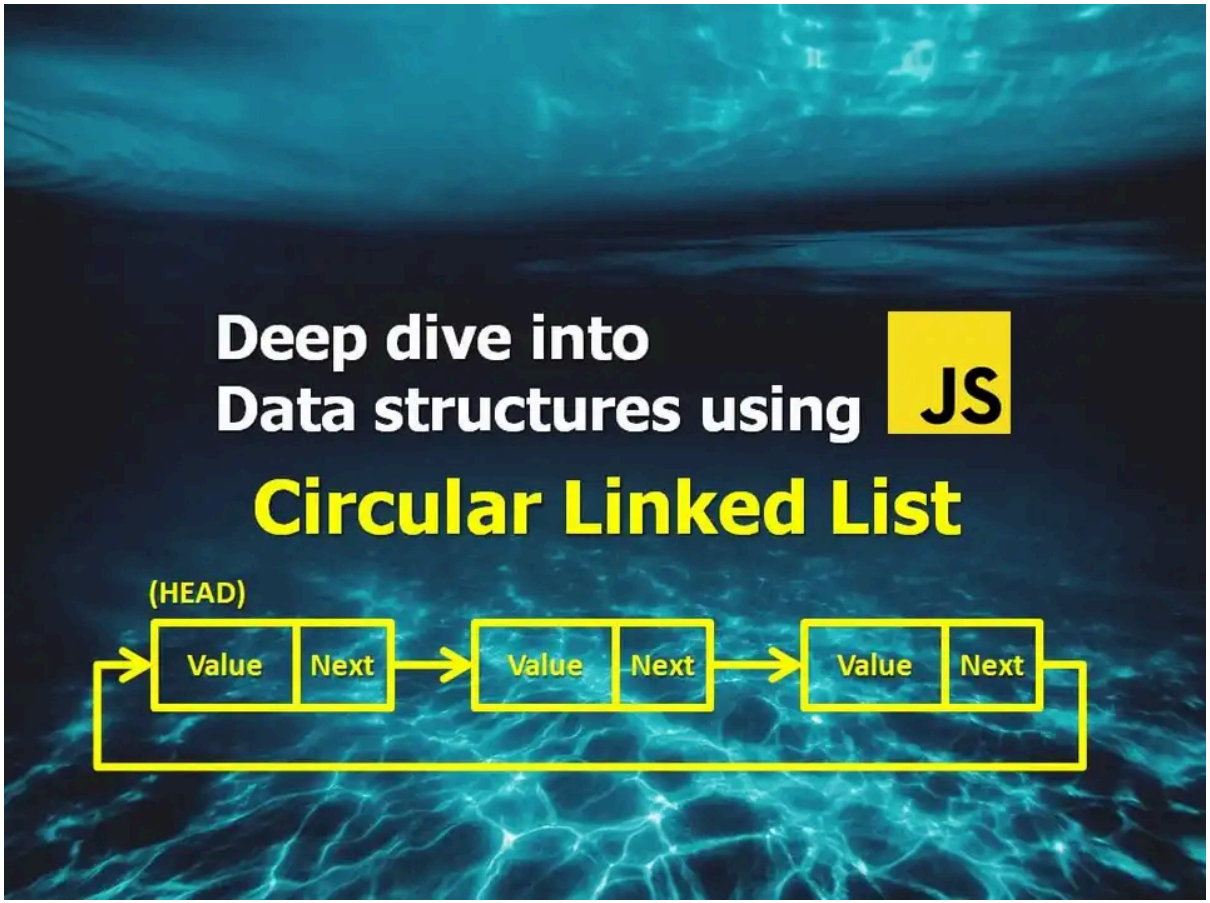
# Deep Dive into Data structures using Javascript - Circular Linked List

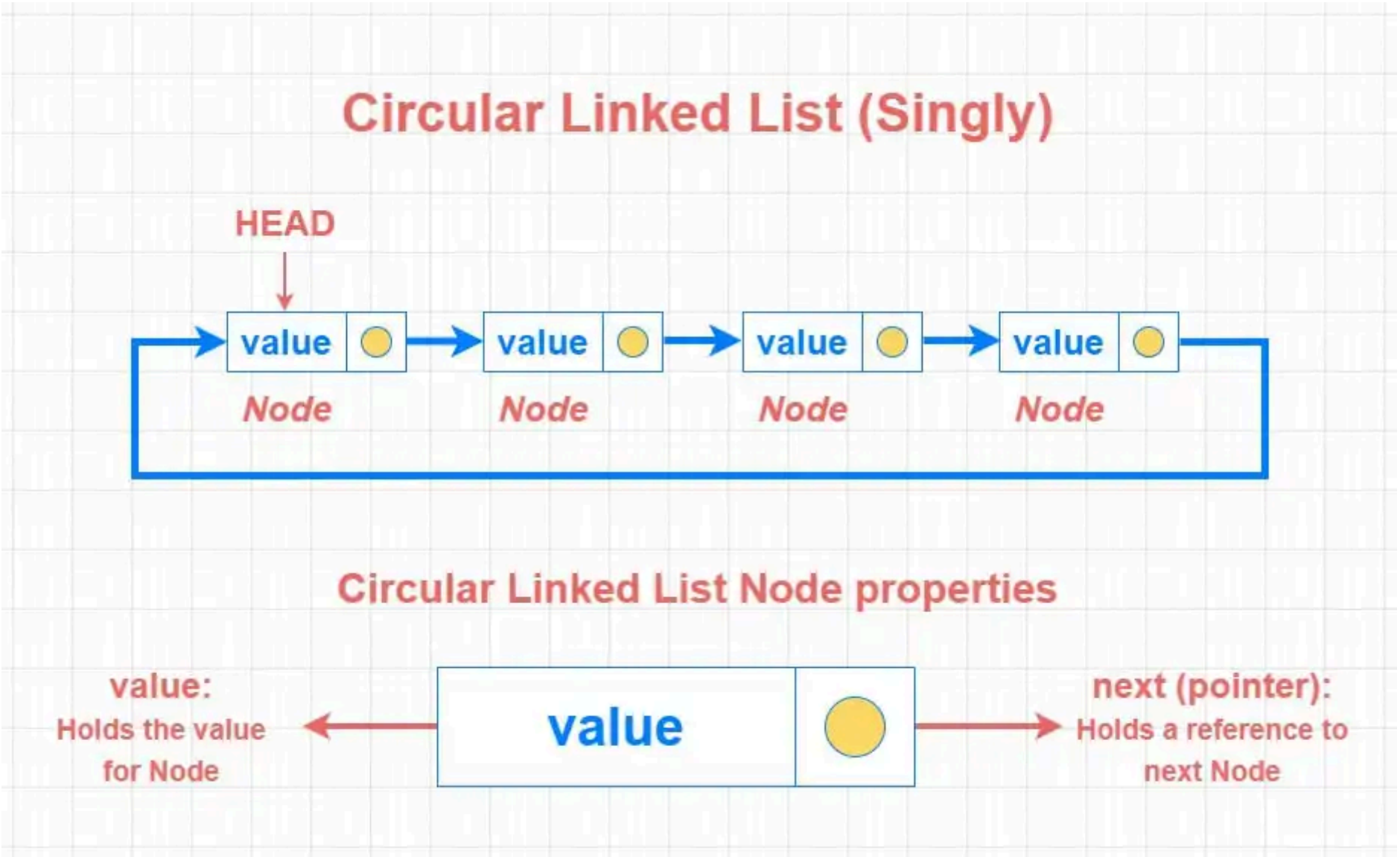



## What is a Circular Linked List?

A Circular Linked list is a variation of Linked List data structure. It contains almost all characteristics of a Linked List with a key difference - the tail and head points to eachother to form a circle of nodes. Which also means we do not have any null pointer at the tail. Because of this structural difference, Circular Linked list classifies as a non-linear data structure unlike the regular Linked list which is linear - since the shape of connected nodes forms a cyclic data structure.

Both Singly and Doubly Linked lists have their Circular variations, but we will be focusing on the Singly Circular Linked List in this article. I will be referring to Singly Linked Lists in some sections, therefore the tone of the article will be assuming you are familiar with the Linked List data structure. If that's not the case or you need a quick refreshment on Linked Lists, I'd suggest you to start from the Linked List article by following the link below, then come back and continue here later:

## Anatomy of a Circular Linked List




A Circular Linked List is consisted by series of connected nodes that forms a cyclic data structure. Each node contains 2 properties:

**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 Linked List is extremely similar with regular 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 Linked List

Circular Linked list shares the same Big O complexity with Linked Lists. We would want to use this whenever we need to deal with operations that has circular logic - a list that accesses items over and over again in a loop. Some real world examples are:

- Music player song list loop feature, whenever the song list ends it starts again from the first song.

- Games using turn based logic like RPGs where it will loop through characters until the level or match ends. Multiplayer games also uses a circular list to swap between different players.

- Operating systems makes use of Circular lists to share CPU time for different users or applications, or processes that will be time shared in Round-Robin order.

- Streaming videos also makes use of this, where the server sends chunks of video and sound with unpredictable rates. Each chunk is added to the list, and we are still able to play the parts that is loaded without having to wait video is fully loaded while it is buffering. If we have a fully loaded video, we can replay it without having to fetch it again.

**Pros:**

- We can traverse the entire list starting from any node.

- Useful and efficient for cases that has circular logic.

- Circular Linked lists can be used to build Circular Queue data structure.

- Circular Linked lists are also used to build advanced data structures like Fibonacci Heaps.

**Cons:**

- It is more complex to implement compared to linear Linked lists.

- If traversal is not handled properly, it can result in an infinite loop.

- If pointers are not handled properly, it can result in losing the list in memory.

## Circular 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 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
    }
}

class SinglyCircularLinkedList {
  constructor(value) {
```

```
    this.head = null
    this.tail = null
    this.length = 0

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

// used to initialize Singly Circular Linked List
initialize(value) {
      // create a node
        const newNode = new Node(value)
        // create a circular reference (points to itself)
      newNode.next = 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)
    // point new node next pointer to this head.
    newNode.next = this.head
    // now, make tails pointer to point to newNode
    this.tail.next = newNode
    // set the tail with 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)
    // point new node next pointer to this head.
    newNode.next = this.head
    // now, make tails next pointer to point to newNode
    this.tail.next = newNode
    // set the head with 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 want 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 and next as null.
  const newNode = new Node(value, null)

  // pick previous index
  const preIdx = this.traverseToIndex(index - 1)
  // pick target index
  const targetIdx = preIdx.next
  // place newNode in front of previous node
```

```
    preIdx.next = newNode
    // place target index in front of new node
    newNode.next = targetIdx
    this.length++
    return this
  }

  deleteHead() {
    // check if there is a head value - if not return a warning (or an error)
    if (this.length === 0) return 'List is empty'
    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
    // now change the head pointer to newHead
    this.head = newHead
    // update tail pointer to point on updated head:
```

```
    this.tail.next = this.head
    this.length--
    return headVal
  }

  deleteTail() {
    // check if length is zero - if not return a warning (or an error)
    if (this.length === 0) return 'List is empty'

    // If there is only one node left
    if (this.length === 1) {
      const headVal = this.head.value
      this.head = null
      this.tail = null
      this.length--
      return headVal
    }

    // Store the current tail value:
    const tailVal = this.tail.value

    // Pick the previous node of tail
    const newTail = this.traverseToIndex(this.length - 2)
    // Make newTail point to the head:
    newTail.next = this.head
    // Make tail to point to newTail, this will remove the tail from the list:
    this.tail.next = newTail
    this.length--
    return tailVal
```

```
}

delete(index) {
  // 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'

  // Handle the case if there is 2 elements left - in this case we either remove head or tail:
  if (this.length === 2) {
    if (index === 0) {
      return this.deleteHead()
    }
    if (index > 0) {
      return this.deleteTail()
    }
  }

  // For a list with more than 2 elements, define removal style.
  // Removal will be either from head, middle or tail.
  let removalType
  if (index === 0) {
    removalType = 'head'
  } else if (index >= this.length) {
    removalType = 'tail'
  } else {
    removalType = 'middle'
  }
```

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

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

    // To remove from middle, we will need both previous and target nodes
    if (removalType === 'middle') {
      const preIdx = this.traverseToIndex(index - 1)
      const targetIdx = preIdx.next
      const targetVal = targetIdx.value
      // Implement removal by pointing preIdx.next to targetIdx.next
      // This will detach the target index node from Linked List
      preIdx.next = targetIdx.next
      this.length--
      return targetVal
    }
  }

  reverse() {
    // Checkup - if list only contains one item, no need to reverse
    if (!this.head.next) return

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

```
    do {
        // Start with taking the next node reference
        nextNode = currentNode.next
        // Then, point the currentNode to previous one
        currentNode.next = previousNode
        // Now, move the previous and current one step forward. How?
        // To move the previousNode one step forward, we reference it to the currentNode:
        previousNode = currentNode
        // To move the currentNode one step forward, we reference it to the nextNode:
        currentNode = nextNode
    } while (currentNode !== this.head)

    this.head.next = previousNode
    this.head = previousNode
    return this.toArray()
  }
}
```