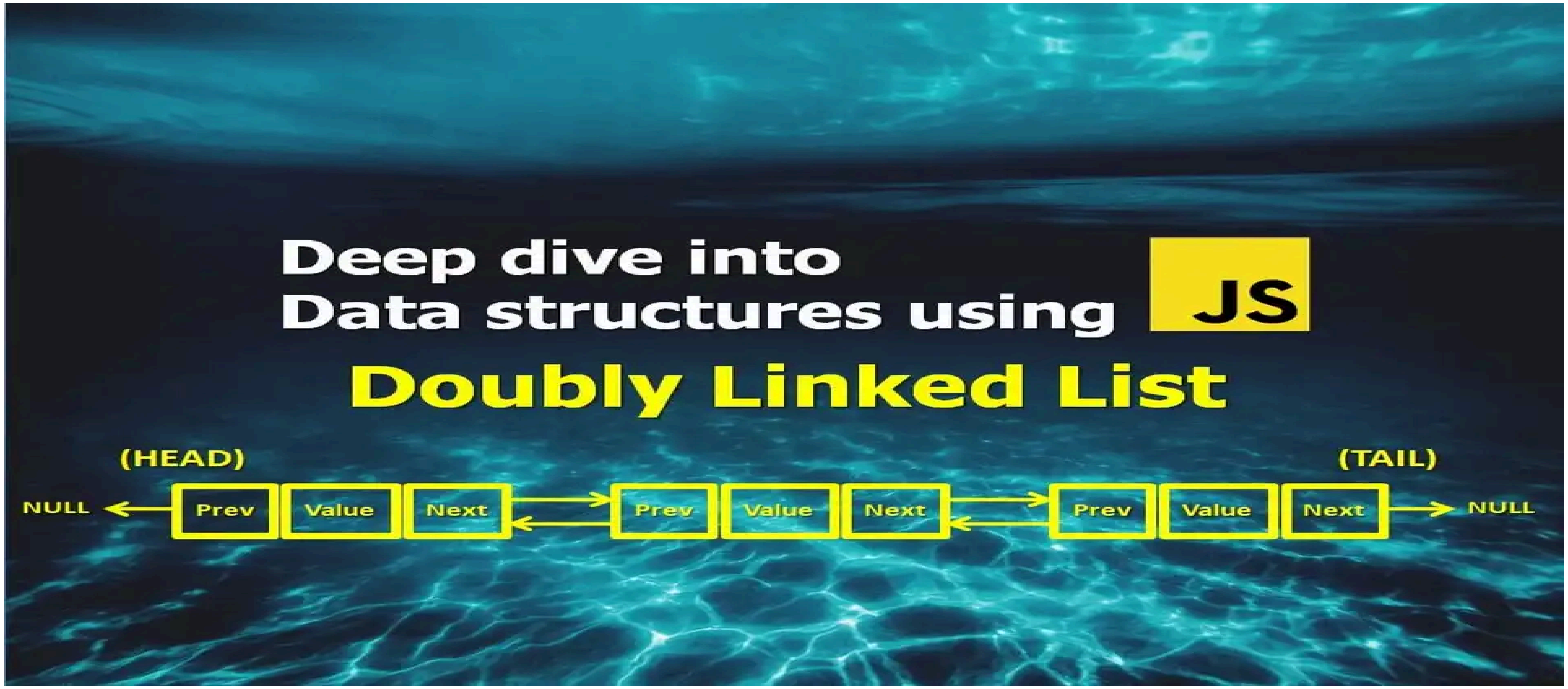
Deep Dive into Data structures using Javascript - Doubly Linked List



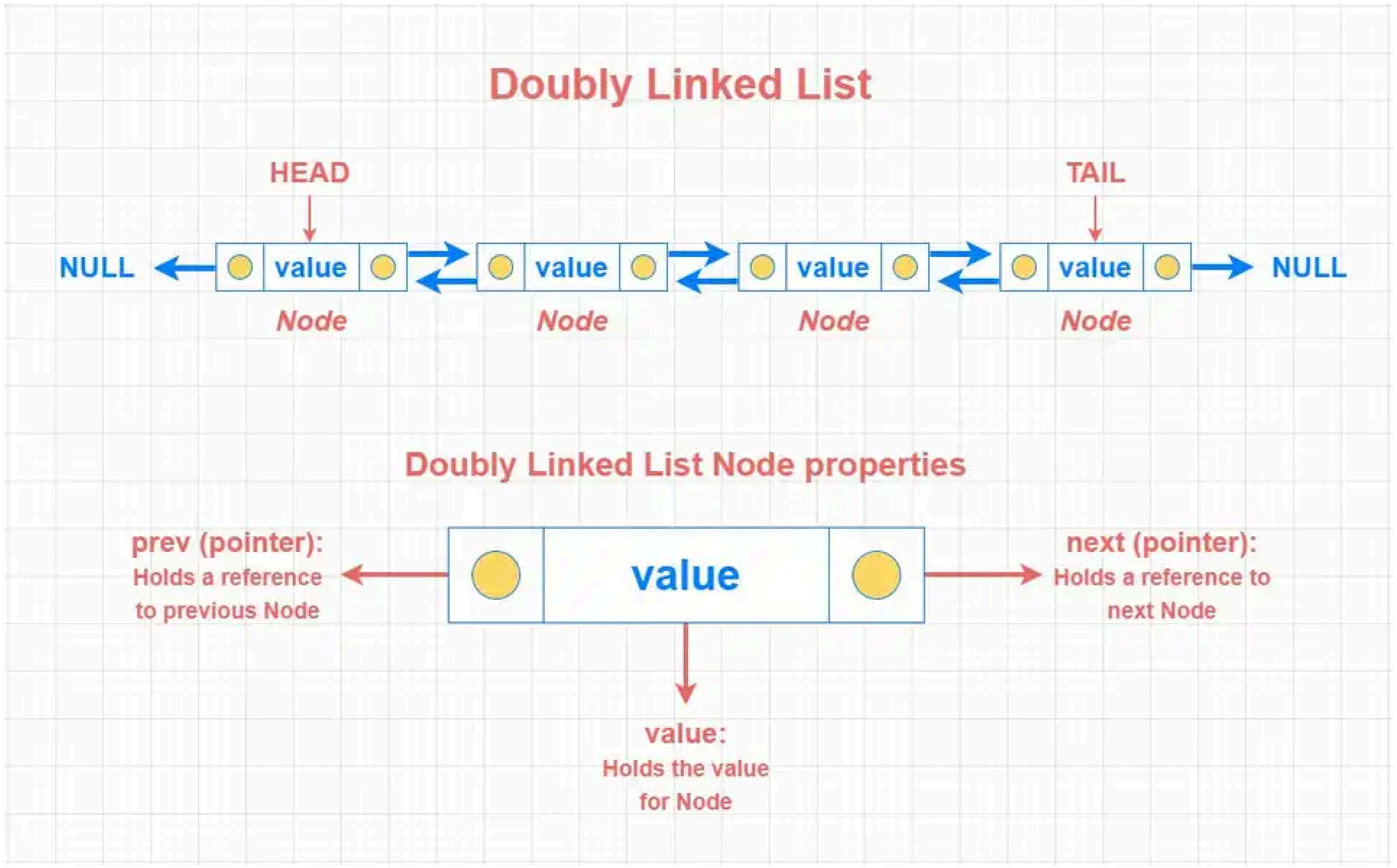
What is a Doubly Linked List?

A Doubly Linked List is a variation of Linked List data structure. It contains all characteristics of a Singly Linked List (or we simply call it Linked List) with one additional feature: each Node contains 2 pointers (previous and next) unlike Singly Linked List which has only one pointer that points to the next Node.

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:

Deep Dive into Data structures using Javascript - Linked List

Anatomy of a Doubly Linked List



A Doubly Linked List is consisted by a series of connected Nodes, each Node contains 3 properties:

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

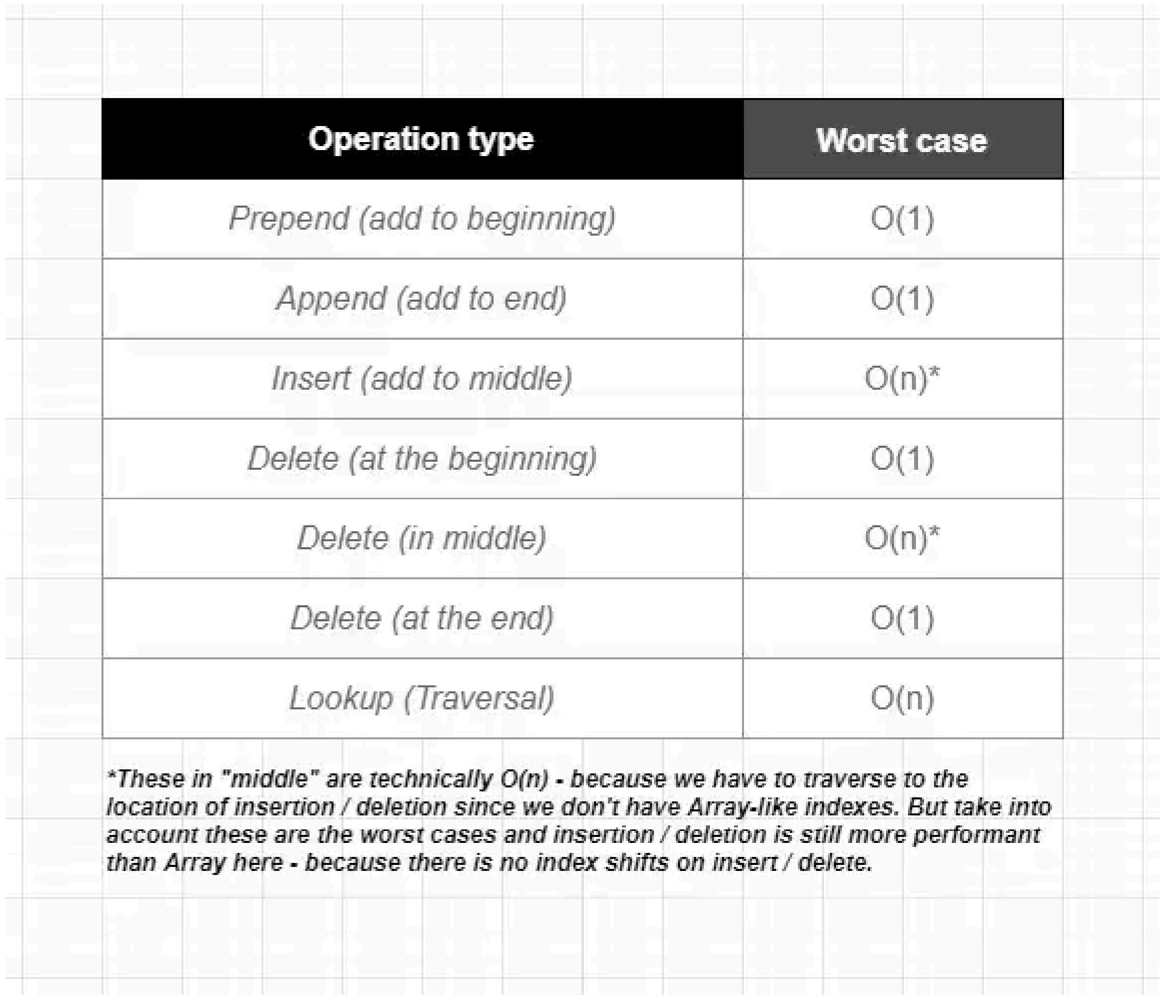
Value: Holds the value / data for the Node.

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

Similar to Singly Linked List, we also call the first node **"HEAD"** and the last node **"TAIL"** here. But you probably have noticed a slight difference with the head node - one part is pointing to null on the visual. Why? It is because since we know the head is always first node - there is no other previous Node in the list to point to. Therefore the previous pointer on the head node will be always pointing to the null.

When and When not to use Doubly Linked List

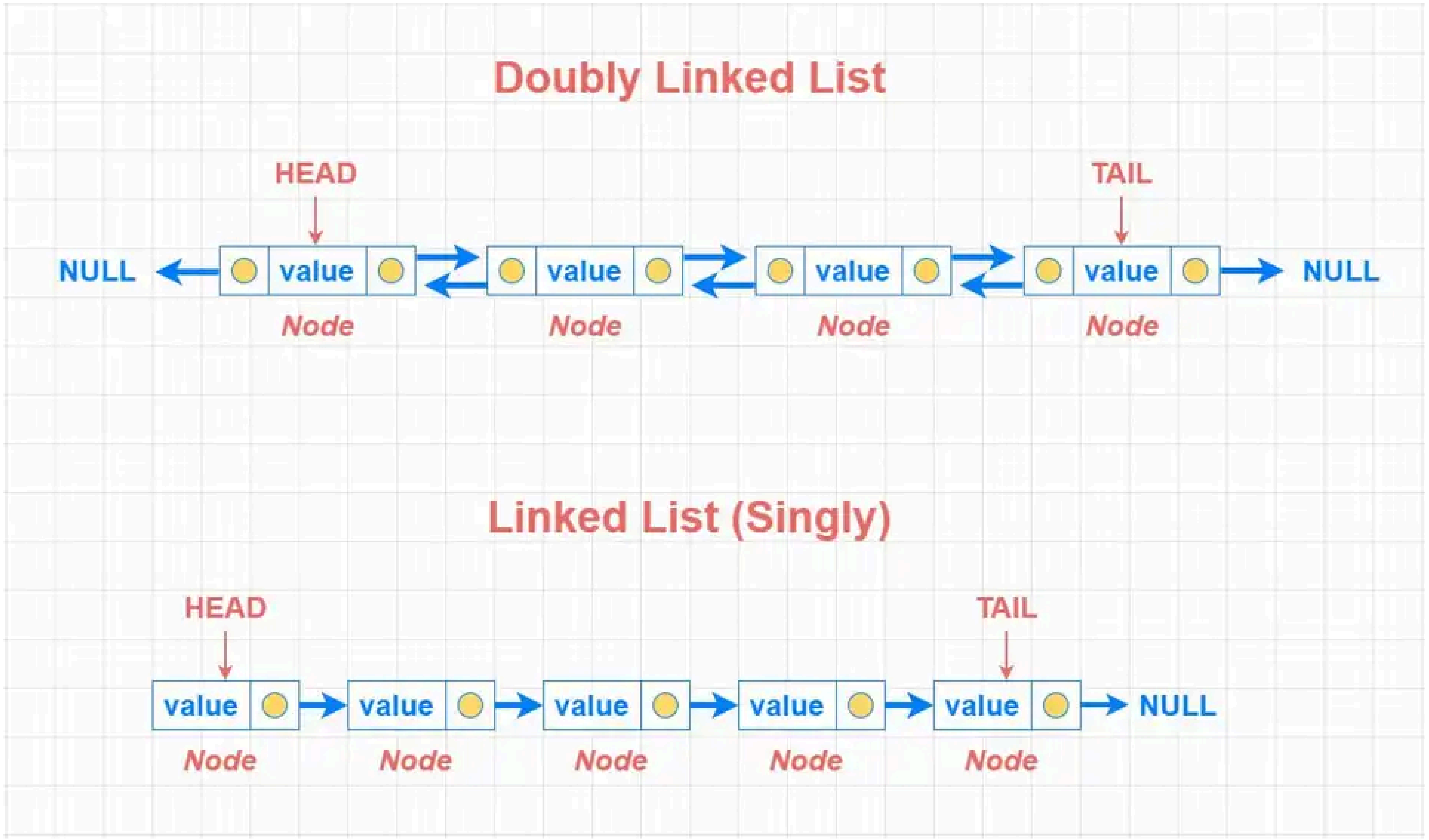
When you have a situation that you might specifically considering to use a Doubly Linked List, it is most likely you have already decided to use a Linked List - and making comparison between if you go with a Singular Linked List or a Doubly Linked List. On next section we will be comparing these two. But let's start with taking a quick look at the Big O of common operations in Doubly Linked List.



Operation type	Worst case
Prepend (add to beginning)	O(1)
Append (add to end)	O(1)
Insert (add to middle)	O(n)*
Delete (at the beginning)	O(1)
Delete (in middle)	O(n)*
Delete (at the end)	O(1)
Lookup (Traversal)	O(n)

**These in "middle" are technically O(n) - because we have to traverse to the location of insertion / deletion since we don't have Array-like indexes. But take into account these are the worst cases and insertion / deletion is still more performant than Array here - because there is no index shifts on insert / delete.*

Doubly Linked List vs Singly Linked List



Whenever we deal with different data structures or their different implementations, answer to what to choose is same: “It depends on the context”. To get a better idea, let’s take a look at pros and cons of each one.

Singly Linked List

Pros:

- Implementation is simpler and more straight-forward compared to Doubly Linked List.
- It requires less memory, due to having a single pointer on each Node.
- Since we need to deal with a single pointer on each Node, there is less operations inside methods.
- Due to having less operations on methods, it operates slightly faster than Doubly Linked List.

Cons:

- Cannot be traversed in reverse direction, because pointers only targets the next Node.
- If the head node is not maintained correctly and lost for some reason, you will lose the rest of the list in memory.

When to use a Singly Linked List

- If you have less memory and memory is expensive.
- Main goal is to do fast insertion & deletion, you don't have to deal with traversal so often.

Doubly Linked List

Pros:

- Better traversal abilities, it can be traversed in both directions (forward or backward).
- deleteTail() method is faster. In Singly Linked List, to remove the tail you need to traverse the whole list until the tail node and this operation takes O(n) Linear time. In Doubly Linked List you can simply use the tail Node's previous pointer - which takes O(1) Constant time.

Cons:

- Implementation is more complex compared to Singly Linked List, due to having 2 pointers to deal with inside the methods.
- Takes more memory space due to having 2 pointers.
- It is slightly slower than Singly Linked List due to more operations needed on pointers inside each method.

When to use a Doubly Linked List

- You don’t have a memory problem.
- You want to do traversals / search elements in the list, ability to traverse backwards will give you better options for optimizing traversal performance.

Doubly Linked List implementation in Javascript

Similar to implementation of Singly Linked List, we will be also using ES6 Classes to build this data structure. If you want, you can open your favorite code editor and follow along with me as we go through the steps.

Step 1 - Build a class for the Doubly Linked List Node

Let’s start with identifying the Node element class, which we can use whenever we need to create a new Node.

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

// Create a new Node:
const newNode = new Node(10)
console.log(newNode)

/* newNode output:
Node {
  value: 10,
  next: null,
  prev: null,
}
*/
```

Step 2 - Build a class for the Doubly Linked List

Now we can go further and create the class for DoublyLinkedList. We know that there should be head and tail properties. For ease of use, we can as well add a length property to keep track of our list length.

Additionally, we can have an option in the constructor to create the Doubly Linked List empty or with a single starter value. We will be looking at the append method at the next step.

```
class DoublyLinkedList {
  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)
    }
  }

const doublyLinkedList = new DoublyLinkedList()
console.log(doublyLinkedList)

/* doublyLinkedList output at initializing stage (empty starter):

DoublyLinkedList {
  head: null,
  tail: null,
  length: 0
}

*/
```

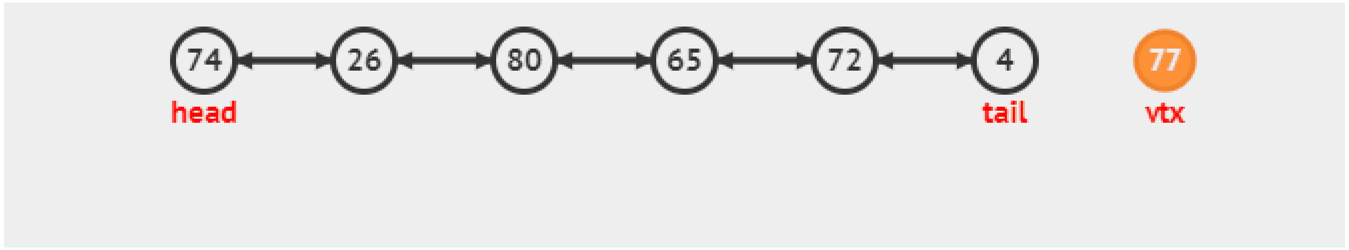
At this point we are done with the base building blocks: `Node` and `DoublyLinkedList` classes. We can continue with extending our DoublyLinkedList class by introducing common methods. To make these methods easier to understand and reason with, I have placed code comments at specific places inside them.

Here is the list of methods we are going to implement:

- `append(value)` - *add to the end*
- `prepend(value)` - *add to the beginning*
- `toArray()` - *return Doubly Linked List elements in an array for ease of debugging*
- `traverseToIndex(index)` - *traversal helper*

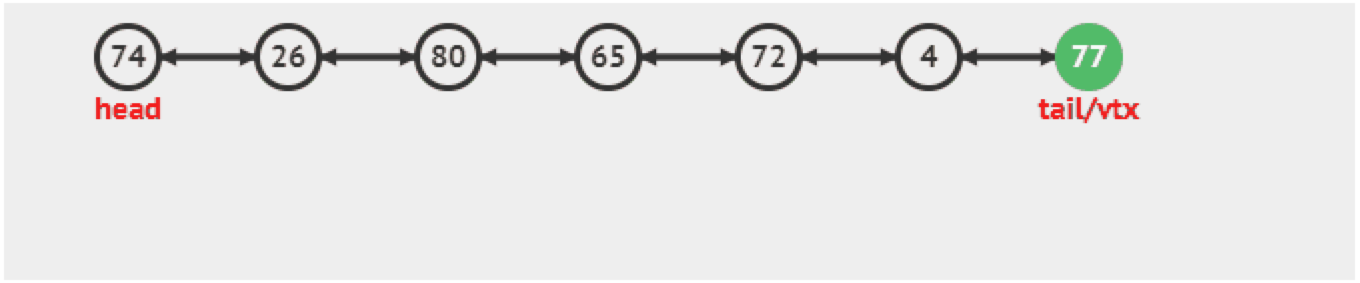
- `insert(index, value)` - *add to the middle*
- `deleteHead()` - *delete from beginning*
- `deleteTail()` - *delete from the end*
- `delete(index)` - *delete from the middle*
- `reverse()` - *reverse order of items*

Step 3 - Doubly Linked List append method



```
// Add to the end of list
append(value) {
    // Initialize a newNode with value recieved
    const newNode = new Node(value)

    // 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
        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 clean
        this.tail = newNode
    }
    this.length++
    return this
}
```

Step 4 - Doubly Linked List prepend method



```
// 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 = new Node(value)
```

```
        // apply a reference to newNode.next prop. When we add it at the start, naturally prepended node's next value should point to the this.
        newNode.next = this.head
        // Since the newNode will be the new previous for the current head, set the prev value of head to be newNode. We do this before changin
        this.head.prev = newNode
        // now that newNode has the this.head as next and newNode as prev, we can set the this.head as newNode directly.
        this.head = newNode
        this.length++
        return this
    }
```

Step 5 - Doubly Linked List toArray method (optional)

To easily debug what is going on our list (or have an option to output Doubly Linked List as an array), we will need toArray method:

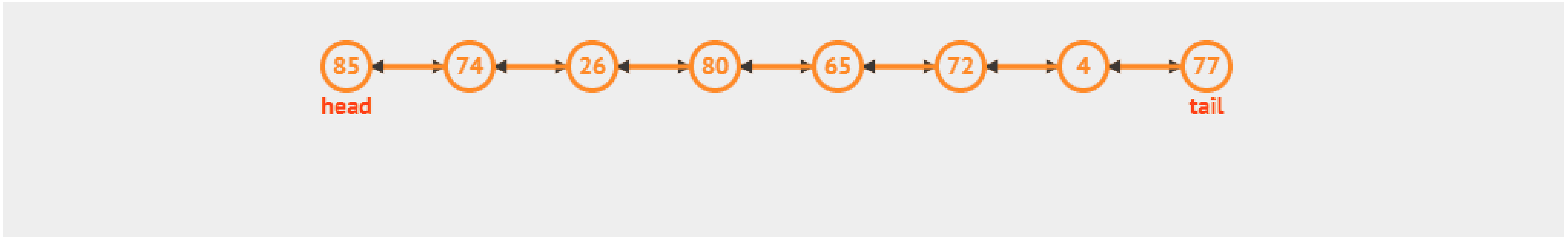
```
    // toArray - loop through nested objects, then return the values in an array
    toArray() {
        const array = []
        let currentNode = this.head

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

Step 6 - Doubly Linked List traverseToIndex method (helper)

Since both insert and removal related methods will have to deal with traversing to a specific index, it will be wise to implement a helper for it:

```
// lookup / traversal helper
traverseToIndex(index) {
    // validate the received index parameter:
    if (!index) return 'Index is missing'
    if (typeof index !== 'number') return 'Index should be a number'

    let counter = 0
    let currentNode = this.head

    while (counter !== index) {
        currentNode = currentNode.next
        counter++
    }

    return currentNode
}
```

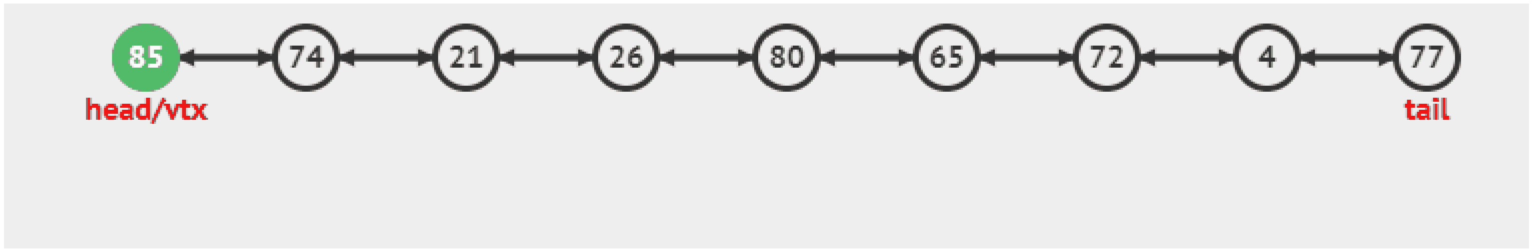
Step 7 - Doubly Linked List insert method



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

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

    // if index is 0, just prepend (add to the beginning)
    if (index === 0) {
        return this.prepend(value)
    }

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

    /*
    Solution flow:
      1 - Pick the previous index Node of target idx
      2 - Pick the target idx Node by using preIdx.next pointer
      3 - Now change previous idx Node pointer to newNode. This will change the previous Node's pointer.
      4 - Now change the newNode.next to targetIdx.
      5 - In other words, we just put the new node in between previous and target: by making previous to point to new node, then new node tc
    */

    // previous one
    const preIdx = this.traverseToIndex(index - 1)
    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!
```

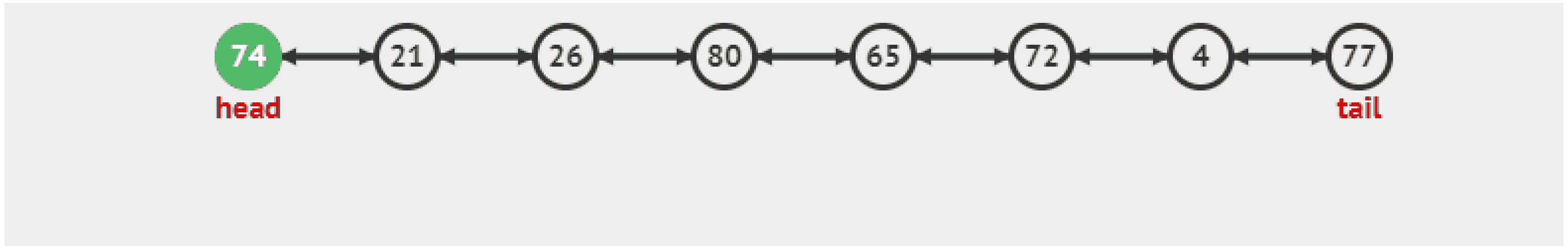
```
    targetIdx.prev = newNode
    this.length++
    return this
}
```

Step 8 - Doubly Linked List deleteHead method



```
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) {
        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
}
```

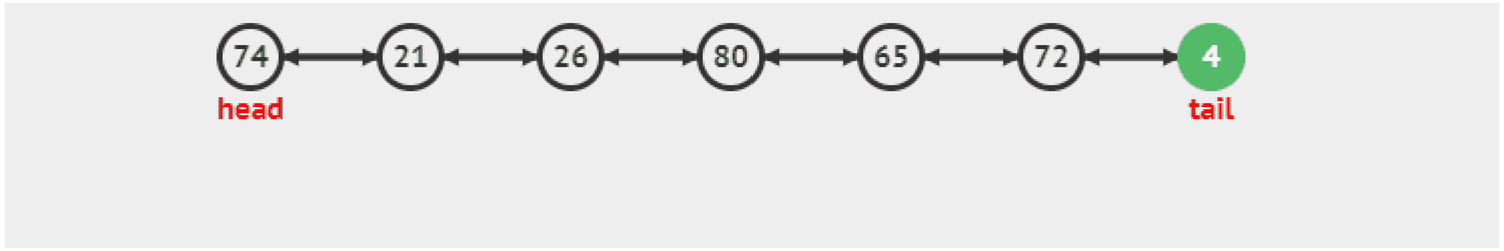
Step 9 - Doubly Linked List deleteTail method



```
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
    }
```

Step 10 - Doubly Linked List delete method



```
delete(index) {
    // 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
```

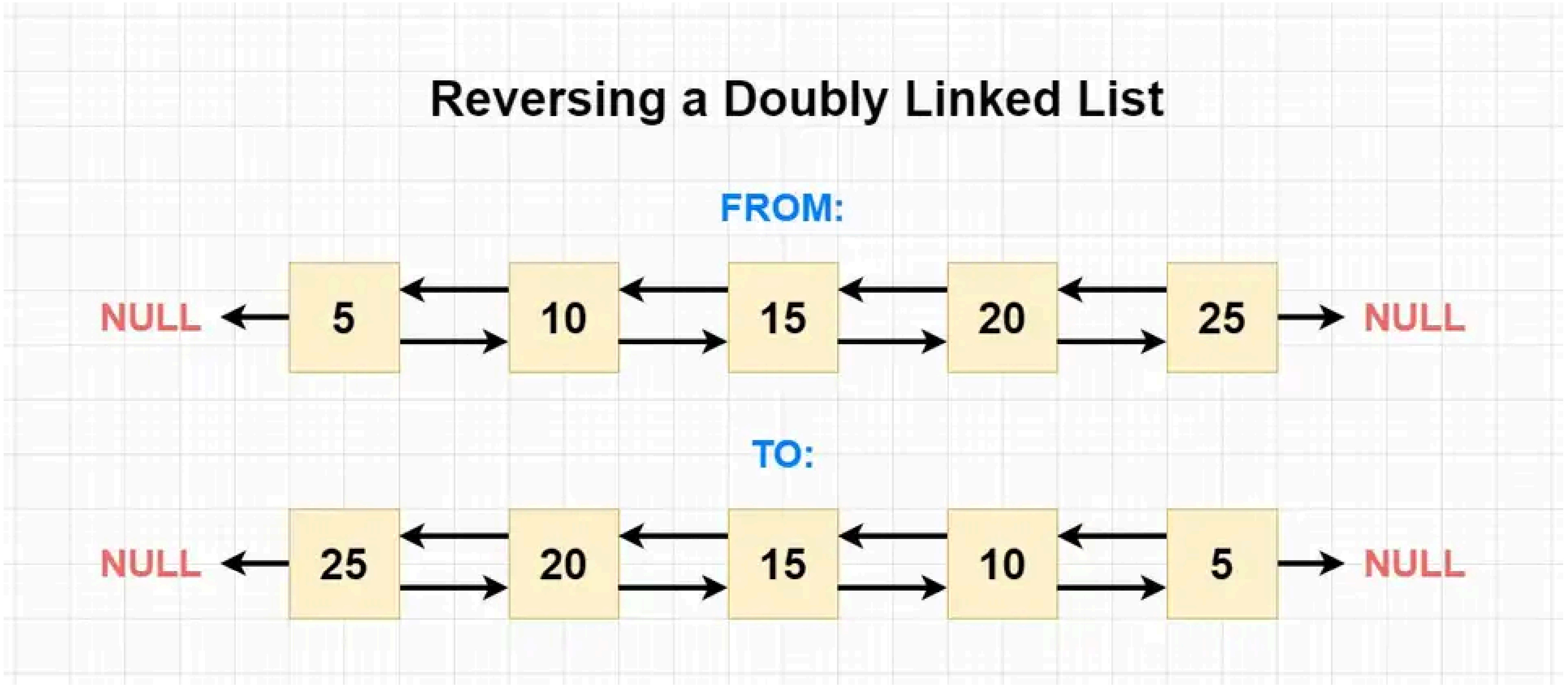
```
if (this.length === 1) {
    this.head = null
    this.tail = null
    this.prev = null
}

// 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 ma
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
    }
}
```

Final step - Reverse a Doubly Linked List

Similar to reversing a Singly Linked List, we will be also using 3 pointers here to reverse the Doubly Linked List. Strategy is very similar with a minor difference - and that is we already have a previous pointer inside each node here. When we deal with reversing a Singly Linked List, we need to create a pointer instead for previous node while traversing since the nodes does not contain them.

Our goal here is basically changing the direction of pointers, one step at a time:



```
reverse() {
  // do not reverse if no elements
    if (this.length === 0) return
  // do not reverse if there is a single element
    if (this.length === 1) return this

    let currNode = this.head
    let prevNode = null
    let nextNode = null

    while (currNode) {
        // Store next node.
```

```
            nextNode = currNode.next
            // Store prev node.
            prevNode = currNode.prev

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

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

        // 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 = prevNode

        return this
    }
```

I hope this article helped you to understand how Doubly Linked Lists works! I'd also like to encourage you to check out this amazing data structures and algorithms visualizer (I have actually generated the gifs you have seen above at this website): https://visualgo.net/en

You can see the full implementation of the Doubly Linked List in Javascript that we went through in this article below. Thanks for reading!

Implementation of Doubly Linked List in Javascript:

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

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

class DoublyLinkedList {
    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 = new Node(value)

        // 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 l
        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 c
        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 = new Node(value)
    // apply a reference to newNode.next prop. When we add it at the start, naturally prepended node's next value should point to the
    newNode.next = this.head
    // Since the newNode will be the new previous for the current head, set the prev value of head to be newNode.
    // We do this before changing the pointer of this.head to newNode. Timing is important!this.head.prev = newNode
    // now that newNode has the this.head as next and newNode as prev, we can set the this.head as newNode directly.
    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
    }

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

        let counter = 0
        let currentNode = this.head

        while (counter !== index) {
            currentNode = currentNode.next
            counter++
        }

        return currentNode
    }

    // insert to specific index
    insert(index, value) {
        // validate the received index parameter:
        if (!index) return 'Index is missing'
        if (typeof index !== 'number') return 'Index should be a number'

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

        // if index is 0, just prepend (add to the beginning)
        if (index === 0) {
            return this.prepend(value)
        }

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

        /*
       Solution flow:
         1 - Pick the previous index Node of target idx
         2 - Pick the target idx Node by using preIdx.next pointer
         3 - Now change previous idx Node pointer to newNode. This will change the previous Node's pointer.
         4 - Now change the newNode.next to targetIdx.
         5 - In other words, we just put the new node in between previous and target: by making previous to point to new node, then new noc
       */

        // previous one
        const preIdx = this.traverseToIndex(index - 1)
        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 ste
        targetIdx.prev = newNode
        this.length++
        return this
    }
```

```
// 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) {
        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
}

// Delete from specific index
delete(index) {
    // 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) {
```

```
        this.head = null
        this.tail = null
        this.prev = null
    }

    // 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 the list
    reverse() {
        // do not reverse if no elements
        if (this.length === 0) return
        // do not reverse if there is a single element
        if (this.length === 1) return this

        let currNode = this.head
        let prevNode = null
        let nextNode = null

        while (currNode) {
            // Store next node.
            nextNode = currNode.next
            // Store prev node.
            prevNode = currNode.prev

            // Change next node of the current node so it would link to previous node.
            currNode.next = prevNode
            currNode.prev = nextNode
            // Move prevNode and currNode nodes one step forward.
            prevNode = currNode
            currNode = nextNode
        }

        // 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 = prevNode

        return this
    }
}
```