Doubly Linked List

Technical Specification: DoublyLinkedList JavaScript Class

1.0 Introduction and Overview

This document provides a formal technical specification for the `DoublyLinkedList` JavaScript class. A doubly linked list is a fundamental data structure in computer science, serving as a highly efficient and flexible tool for managing ordered collections of data. Its strategic importance lies in its ability to support rapid insertion and deletion operations at any point in a sequence.

The `DoublyLinkedList` is defined as a linear data structure composed of a sequence of interconnected nodes. The primary advantage of a doubly linked list over its singly linked counterpart is its capacity for efficient bidirectional traversal. This means the list can be navigated both forwards (from head to tail) and backwards (from tail to head) with equal ease. This capability is made possible because each node within the list maintains two distinct pointers: one referencing the `next` node in the sequence and another referencing the `prev` (previous) node.

The foundational component of this entire structure is the `Node` class, which encapsulates the data and the linkage pointers for each element in the list.

2.0 The Class Specification

The `Node` class is the core building block of the `DoublyLinkedList`. Each instance of the `Node` class serves as a container, holding a single data element along with the critical linkage pointers that connect it to its neighbors, thereby forming the chain that constitutes the list.

2.1 Purpose

The `Node` class's primary role is to encapsulate user-provided data along with the `prev` and `next` references that connect it to adjacent nodes within the `DoublyLinkedList`.

2.2 Properties

A `Node` instance contains the following properties:

Property	Type	Description
`data`	`any`	The value or payload stored within the node.
`prev`	`Node | null`	A reference to the preceding node in the list. The default value is `null`.
`next`	`Node | null`	A reference to the succeeding node in the list. The default value is `null`.

2.3 Constructor:

The constructor initializes a new `Node` object using ES6 default parameters. It accepts `data` as a mandatory argument. The `prev` and `next` parameters are assigned `null` by default if not explicitly provided during instantiation, allowing for the creation of an initially disconnected node.

With the specification of the individual `Node` established, we can now define the `DoublyLinkedList` class, which orchestrates these nodes into a functional, high-level data structure.

3.0 The Class Specification

The `DoublyLinkedList` class acts as the high-level manager for the collection of `Node` objects. It encapsulates all the logic for list manipulation and provides the public interface through which all list operations are performed.

3.1 Purpose

The purpose of the `DoublyLinkedList` class is to provide a complete and robust API for creating and managing the lifecycle of a doubly linked list. This includes all common operations for modification (e.g., adding, removing elements), access (e.g., checking size), and traversal.

3.2 Properties

An instance of the `DoublyLinkedList` class maintains the following essential properties:

Property	Type	Description
`head`	`Node | null`	A reference to the first node in the list. For an empty list, its value is `null`.
`tail`	`Node | null`	A reference to the last node in the list. This property serves a dual strategic purpose: it enables O(1) time complexity for append operations by providing

		direct access to the end of the list, and it serves as the entry point for efficient backward traversal from tail to head. For an empty list, its value is `null`.
`size`	`number`	An integer representing the current number of nodes in the list.

3.3 Constructor:

The class features a parameter-less constructor. When invoked, it initializes a new, empty `DoublyLinkedList` instance by setting the `head` and `tail` properties to `null` and the `size` property to `0`.

Having defined the structure of the list itself, the following section provides a detailed breakdown of the methods that form its public API.

4.0 Class Methods API

This section provides a comprehensive specification for each public method available on the `DoublyLinkedList` class. The methods are organized into logical categories to enhance clarity: **Modification Methods**, **Utility & Accessor Methods**, and **Traversal & Display Methods**.

4.1 Modification Methods

4.1.1

- **Purpose:** Adds a new node containing the provided data to the end of the list.
- **Parameters:**

Name	Type	Description
`data`	`any`	The data to be stored in the new node.

- **Returns:** `void`
- **Logic:** This operation achieves O(1) time complexity by directly accessing the end of the list via the `tail` property, avoiding a full list traversal.
  1. A new `Node` is created with the given `data`. Its `prev` pointer is set to the current `tail` of the list.
  2. An edge case is checked: if the list is empty (`head` is `null`), both the `head` and `tail` properties are set to point to this new node.

3. If the list is not empty, the `next` pointer of the current `tail` node is updated to point to the new node.
4. The list's `tail` property is updated to reference the new node, making it the new last element.
5. The `size` of the list is incremented by one.

4.1.2

- **Purpose:** Adds a new node containing the provided data to the beginning of the list.
- **Parameters:**

Name	Type	Description
`data`	`any`	The data to be stored in the new node.

- **Returns:** `void`
- **Logic:**
  1. A new `Node` is created with the given `data`. Its `next` pointer is set to the current `head` of the list.
  2. An edge case is checked: if the list is empty, both the `head` and `tail` properties are set to point to this new node.
  3. If the list is not empty, the `prev` pointer of the current `head` node is updated to point to the new node.
  4. The list's `head` property is updated to reference the new node, making it the new first element.
  5. The `size` of the list is incremented by one.

4.1.3

- **Purpose:** Inserts a new node with the given data at a specified zero-based index.
- **Parameters:**

Name	Type	Description
`data`	`any`	The data for the new node.
`index`	`number`	The zero-based position at which to insert the node.

- **Returns:** `boolean` — Returns `true` on successful insertion and `false` if the index is invalid.
- **Logic:**
  1. The `index` is validated. If it is less than `0` or greater than the current `size`, an error message is logged to the console via `console.error` and the method returns `false`.
  2. Handles boundary conditions:
     - If `index` is `0`, it delegates the operation to `prepend(data)` and returns `true`.
     - If `index` is equal to `size`, it delegates the operation to `append(data)` and returns `true`.
  3. For any other valid index, the list is traversed from the `head` until the node at the target `index` is located.
  4. A new `Node` is created. Its `prev` pointer is set to the previous node of the target location, and its `next` pointer is set to the node currently at the target location.
  5. The surrounding pointers are re-linked: the `next` pointer of the preceding node and the `prev` pointer of the succeeding node are both updated to point to the new node.
  6. The `size` is incremented, and the method returns `true`.

4.1.4

- **Purpose:** Removes the node at a specified zero-based index and returns its data.
- **Parameters:**

Name	Type	Description
`index`	`number`	The zero-based position of the node to remove.

- **Returns:** `any | null` — Returns the data of the removed node, or `null` if the index is invalid.
- **Logic:**
  1. The `index` is validated. If it is less than `0` or greater than or equal to `size`, the method returns `null`.

2. The removal logic is handled based on the node's position:
    - **First Node (`index === 0`):** The `head`'s data is stored. The list's `head` is advanced to the second node. If the new `head` exists, its `prev` pointer is set to `null`. If the list becomes empty, the `tail` is also set to `null`.
    - **Last Node (`index === size - 1`):** The `tail`'s data is stored. The list's `tail` is moved to the second-to-last node. The new `tail`'s `next` pointer is set to `null`.
    - **Middle Node:** The list is traversed to find the target node, and its data is stored. To remove the node, the `next` pointer of the target node's preceding node is re-linked to the target's succeeding node. Symmetrically, the `prev` pointer of the succeeding node is re-linked to the preceding node, effectively bypassing and de-linking the target node from the list.
3. After removal, the `size` of the list is decremented.
4. The stored data from the removed node is returned.

4.2 Utility & Accessor Methods

4.2.1 isEmpty()

- **Purpose:** Checks if the linked list is empty.
- **Parameters:** This method does not accept any parameters.
- **Returns:** `boolean` — Returns `true` if the list contains no nodes, `false` otherwise.
- **Logic:** The method evaluates the expression `this.size === 0` and returns the result.

4.2.2 size()

- **Purpose:** Returns the current number of nodes in the list.
- **Parameters:** This method does not accept any parameters.
- **Returns:** `number` — The integer value of the `size` property.
- **Logic:** The method directly returns the value of `this.size`.

4.3 Traversal & Display Methods

4.3.1 printForward()

- **Purpose:** Logs a formatted string representation of the list to the console, traversing from head to tail.
- **Parameters:** This method does not accept any parameters.
- **Returns:** `void`
- **Logic:**
    1. A temporary pointer, `current`, is initialized to `this.head`.
    2. The method iterates through the list, following the `next` pointers, until `current` becomes `null`.
    3. In each iteration, the `data` of the `current` node is appended to a result string.
    4. Once traversal is complete, the final formatted string is logged to the console.

4.3.2 printBackward()

- **Purpose:** Logs a formatted string representation of the list to the console, traversing from tail to head.
- **Parameters:** This method does not accept any parameters.
- **Returns:** `void`
- **Logic:**
    1. A temporary pointer, `current`, is initialized to `this.tail`.
    2. The method iterates through the list in reverse, following the `prev` pointers, until `current` becomes `null`.
    3. In each iteration, the `data` of the `current` node is appended to a result string.
    4. Once traversal is complete, the final formatted string is logged to the console.

Having specified the complete API, the final section provides a practical demonstration of its usage.

5.0 Implementation and Usage Example

This final section provides a concrete code example to demonstrate the instantiation and practical application of the `DoublyLinkedList` class and its methods. This grounds the theoretical specification in a real-world use case, illustrating how the various API methods work together.

5.1 Sample Code

```
const list = new DoublyLinkedList();

// Append values
list.append(10);
list.append(20);
list.append(30);
list.append(40);

// Prepend a value
list.prepend(50);

// Insert a value at a specific index
list.insertAt(60, 2);

// Print the list forwards and backwards
list.printListForward();
list.printListBackward();
```

5.2 Expected Output

Running the sample code above will produce the following output in the console, demonstrating the final state of the list after all modification operations have been completed.

```
50 ↔ 10 ↔ 60 ↔ 20 ↔ 30 ↔ 40 ↔ null
40 ↔ 30 ↔ 20 ↔ 60 ↔ 10 ↔ 50 ↔ null
```

This specification defines the `DoublyLinkedList` class as a well-defined and feature-complete data structure implementation, suitable for managing ordered data collections with high efficiency.

Implementing a Doubly Linked List in JavaScript: A Comprehensive Guide

A doubly linked list is a fundamental linear data structure where elements are stored in a sequence, but unlike arrays, they are not in contiguous memory locations. Its primary advantage over its simpler counterpart, the singly linked list, is the ability for bidirectional traversal. Each element, or "node," in a doubly linked list contains pointers to both the next and the previous node in the sequence. This guide provides a complete, step-by-step implementation of a doubly linked list in JavaScript, covering the construction of the core `Node` and `DoublyLinkedList` classes and all essential methods for manipulation and traversal.

This guide will walk you through building the key components from the ground up: the `Node` class, which serves as the basic building block; the `DoublyLinkedList` class, which manages the collection of nodes using `head`, `tail`, and `size` properties; and a full suite of methods for insertion, deletion, and traversal.

1. The Building Block: The Class

The atomic unit of any linked list is the `Node`. Each node is an object that encapsulates a piece of data and the connections required to form the chain-like structure of the list. In a doubly linked list, every node contains three essential components: a data payload and two pointers, `prev` and `next`, which establish the bidirectional links that define the data structure.

```
class Node {
  constructor(data, prev = null, next = null) {
    this.data = data;
    this.prev = prev;
    this.next = next;
  }
}
```

The `constructor` initializes a new node. It accepts the `data` to be stored as its primary argument. Crucially, it also accepts arguments for the `prev` and `next` pointers, which are strategically set to `null` by default. This use of default arguments simplifies the creation of new nodes, as their pointers are often unknown or intentionally `null` upon initial creation (e.g., when a new node is the last in the list, its `next` pointer will be `null`).

With the individual `Node` component defined, we now need a larger class to manage a collection of these nodes and orchestrate their interactions.

2. The Container: The Class

The `DoublyLinkedList` class acts as the high-level manager for the entire data structure. It encapsulates the core logic for adding, removing, and finding nodes, providing a clean, user-facing API for interacting with the list without needing to manage pointers directly.

```
class DoublyLinkedList {
    constructor() {
        this.head = null;
        this.tail = null;
        this.size = 0;
    }
}
```

The constructor initializes the list in an empty state with three critical properties:

- **`this.head`**: This property acts as the entry point to the list, always pointing to the very first node. When the list is empty, `head` is `null`.
- **`this.tail`**: This property maintains a direct pointer to the last node in the list. This is a key optimization that allows for appending new nodes and initiating backward traversal in constant time, O(1)—a significant advantage over a singly linked list (or one without a tail pointer), where appending would require an O(n) traversal from the head to find the last node.
- **`this.size`**: This property maintains a real-time count of the nodes currently in the list. Keeping track of the size directly provides an efficient O(1) way to check the list's length, avoiding the need to iterate through all nodes to count them.

Now that the foundational structure is in place, we can begin adding functionality, starting with the core methods for inserting elements into the list.

3. Adding Elements: Core Insertion Methods

Insertion methods are fundamental to making any data structure useful. They provide the mechanism to populate the list with data. This section will cover the three primary ways to add new nodes: to the end of the list (`append`), to the beginning (`prepend`), and at a specific position within the list (`insertAt`).

3.1. Appending a Node to the End

`append` adds a new node to the very end of the list. The process involves creating a new node and ensuring its `prev` pointer is correctly linked to the current `tail` of the list.

```
append(data) {
  const newNode = new Node(data, this.tail);

  if (!this.head) {
    this.head = newNode;
    this.tail = newNode;
  } else {
    this.tail.next = newNode;
    this.tail = newNode;
  }
  this.size++;
}
```

First, a new node is created. Its `data` is set, and its `prev` pointer is immediately linked to the current `this.tail`. This line is strategically placed before the conditional logic because the new node's `prev` pointer will *always* be set to the current `this.tail`, regardless of whether the list is empty (`this.tail` is `null`) or not. This is an elegant and efficient approach. The logic then splits based on whether the list is empty:

- **Empty List (`if` block):** If `this.head` is `null`, the list is empty. In this case, the new node is the only node, so both `this.head` and `this.tail` must be updated to point to it.
- **Non-Empty List (`else` block):** If the list already contains nodes, the `next` pointer of the current `tail` is updated to point to the `newNode`. Finally, `this.tail` is updated to reference the `newNode`, officially making it the new end of the list.

3.2. Prepending a Node to the Beginning

`prepend` adds a new node to the start of the list, making it the new `head`. This involves creating a new node whose `next` pointer links to the current `head`.

```
prepend(data) {
  const newNode = new Node(data, null, this.head);

  if (!this.head) {
    this.head = newNode;
    this.tail = newNode;
  } else {
    this.head.prev = newNode;
    this.head = newNode;
  }
  this.size++;
}
```

A `newNode` is created with its `next` pointer correctly set to the current `this.head`. Its `prev` pointer is `null`, as it will be the new first node. The logic again handles two distinct scenarios:

- **Empty List (`if` block):** As with `append`, if the list is empty, both `head` and `tail` are set to point to the `newNode`.
- **Non-Empty List (`else` block):** For a non-empty list, the `prev` pointer of the old `head` is updated to link back to the `newNode`. Then, `this.head` is updated to reference the `newNode`, making it the new start of the list.

3.3. Inserting a Node at a Specific Index

While adding to the ends is common, `insertAt` provides the flexibility to add a node anywhere in the list. This operation requires careful handling of edge cases and precise pointer manipulation for insertions in the middle.

```
insertAt(data, index) {
  if (index < 0 || index > this.size) {
    console.error("Invalid index");
    return false;
  }
  if (index === 0) {
    this.prepend(data);
```

```
        return true;
    }
    if (index === this.size) {
        this.append(data);
        return true;
    }

    let current = this.head;
    let count = 0;
    while (count < index) {
        current = current.next;
        count++;
    }

    const newNode = new Node(data, current.prev, current);
    current.prev.next = newNode;
    current.prev = newNode;
    this.size++;
    return true;
}
```

This method is composed of several logical blocks:

1. **Index Validation:** The initial `if` statement acts as a guard clause, checking for out-of-bounds indices. An index is valid if it's between `0` and `this.size` (inclusive, as inserting at `this.size` is equivalent to appending).
2. **Edge Case Optimization:** The code efficiently handles insertions at the beginning (`index === 0`) and end (`index === this.size`) by reusing the highly optimized `prepend` and `append` methods, respectively.
3. **Traversal:** For insertions in the middle, a `while` loop traverses the list. The `current` pointer starts at the `head` and moves forward until it points to the node currently at the target `index`.
4. **Pointer Manipulation:** This is the core of the insertion logic. Once the target location is found, the `newNode` is surgically inserted with three precise pointer updates:

    a. **Node Creation:** `const newNode = new Node(data, current.prev, current);` correctly establishes the new node's own `prev` and `next` pointers, linking it to its future neighbors.

b. **Updating the Predecessor:** `current.prev.next = newNode;` re-routes the `next` pointer of the node *before* the insertion point to point to our new node.

c. **Updating the Successor:** `current.prev = newNode;` re-routes the `prev` pointer of the original node *at* the insertion point to point back to our new node, completing the insertion.

4. Removing Elements: The Method

Removing a node, the inverse of insertion, requires equally careful pointer manipulation to maintain the list's integrity. The `removeAt` method is designed to handle three distinct cases: removing the `head`, removing the `tail`, and removing a node from the middle of the list.

```
removeAt(index) {
    if (index < 0 || index >= this.size) {
        return null;
    }

    let removedData;

    if (index === 0) { // Removing the head
        removedData = this.head.data;
        this.head = this.head.next;
        if (this.head) {
            this.head.prev = null;
        } else {
            this.tail = null; // List is now empty
        }
    } else if (index === this.size - 1) { // Removing the tail
        let current = this.tail;
        removedData = current.data;
        this.tail = current.prev;
        this.tail.next = null;
    } else { // Removing from the middle
        let current = this.head;
        let count = 0;
        while (count < index) {
            current = current.next;
            count++;
        }
        removedData = current.data;
        current.prev.next = current.next;
        current.next.prev = current.prev;
    }

    this.size--;
    return removedData;
}
```

The method's logic is partitioned to handle each removal scenario with precision:

- **Index Validation:** The first step is to ensure the provided `index` is valid (from `0` to `size - 1`). If not, `null` is returned to indicate failure.
- **Removing the Head (`index === 0`):** The `head` pointer is simply moved to the next node in the list (`this.head.next`). A critical check follows: if the list is not empty after removal, the new `head`'s `prev` pointer must be set to `null`. If the removal empties the list, the `tail` must also be set to `null`.
- **Removing the Tail (`index === this.size - 1`):** The `tail` pointer is moved to the previous node (`this.tail.prev`). The new `tail`'s `next` pointer is then severed by setting it to `null`, officially making it the new end of the list.
- **Removing a Middle Node (`else` block):** The list is traversed to find the node targeted for removal. This operation removes the `current` node from the chain by linking its predecessor directly to its successor (`current.prev.next = current.next`) and its successor back to its predecessor (`current.next.prev = current.prev`).

After successfully removing the node, the `size` is decremented, and the data of the removed node is returned.

5. Utility and Traversal Methods

A robust data structure class includes utility methods for status checks and traversal methods for inspecting its contents. This section covers simple helpers like `getSize` and `isEmpty`, and more importantly, methods for printing the list's contents both forward and backward, which highlights the primary feature of a doubly linked list.

```
getSize() {
    return this.size;
}

isEmpty() {
    return this.size === 0;
}

printListForward() {
    let current = this.head;
    let result = '';
    while (current) {
        result += `← ${current.data} → `;
        current = current.next;
    }
    console.log(result + 'null');
}

printListBackward() {
    let current = this.tail;
    let result = '';
    while (current) {
        result += `← ${current.data} → `;
        current = current.prev;
    }
    console.log('null ' + result);
}
```

- **`getSize()` and `isEmpty()`**: These are simple and highly efficient O(1) helper functions. They directly return the value of the `size` property or a boolean derived from it, providing instant status information about the list.
- **`printListForward()`**: This method demonstrates standard list traversal. It initializes a `current` pointer at `this.head` and iterates through the list by repeatedly moving to the next node (`current = current.next`) until it reaches the end (`null`), building a result string along the way.

- **`printListBackward()`**: This method perfectly demonstrates the power of the `tail` pointer and the `prev` links. By starting at `this.tail` and traversing backward using `current = current.prev`, we achieve efficient reverse traversal—a key advantage over singly linked lists.

6. Putting It All Together: A Practical Example

The final step in building a data structure is to test its functionality to ensure it behaves as expected. The following code instantiates the `DoublyLinkedList` and uses the methods we've implemented to build, modify, and print a list, confirming that the logic is sound.

```
// Test Code
const list = new DoublyLinkedList();
list.append(10);
list.append(20);
list.append(30);
list.append(40);
list.prepend(5);
list.insertAt(60, 2);

console.log('Forward Traversal:');
list.printListForward();

console.log('Backward Traversal:');
list.printListBackward();
```

By tracing the sequence of operations in the test code, we can determine the final state of the list:

1. **`append(10, 20, 30, 40)`**: The list becomes `10 → 20 → 30 → 40`.
2. **`prepend(5)`**: A `5` is added to the beginning, resulting in `5 → 10 → 20 → 30 → 40`.
3. **`insertAt(60, 2)`**: A `60` is inserted at index 2. This means the new node containing `60` will become the new element at index 2, pushing the original node (`20`) and all subsequent nodes one position to the right. The final list is `5 → 10 → 60 → 20 → 30 → 40`.

Running the test code will produce the following output in the console:

```
Forward Traversal:
← 5 → ← 10 → ← 60 → ← 20 → ← 30 → ← 40 → null
Backward Traversal:
null ← 40 → ← 30 → ← 20 → ← 60 → ← 10 → ← 5 →
```