**Technical Specification Document: Doubly Linked List Data Structure**

**1. Introduction**

A **Doubly Linked List (DLL)** is a type of linked list where each node contains data and links to both the previous and next nodes. This structure is used for applications requiring bi-directional traversal and efficient insertions or deletions without reallocation, which makes DLLs ideal for managing sequences of data in real-time applications like navigation systems, media players, and undo/redo functionalities.

**2. Use Cases**

Some common use cases of a doubly linked list include:

* **Browser History Navigation**: To navigate back and forth through visited web pages.
* **Text Editor**: For managing undo/redo actions by tracking previous and next states.
* **Playlist Management**: For media players that allow users to navigate through songs or videos.

**3. Data Structure Definition**

In a doubly linked list, each **Node** has:

* **Data**: The value or data payload of the node.
* **Prev Pointer**: Points to the previous node in the list.
* **Next Pointer**: Points to the next node in the list.

Each node is linked bi-directionally:

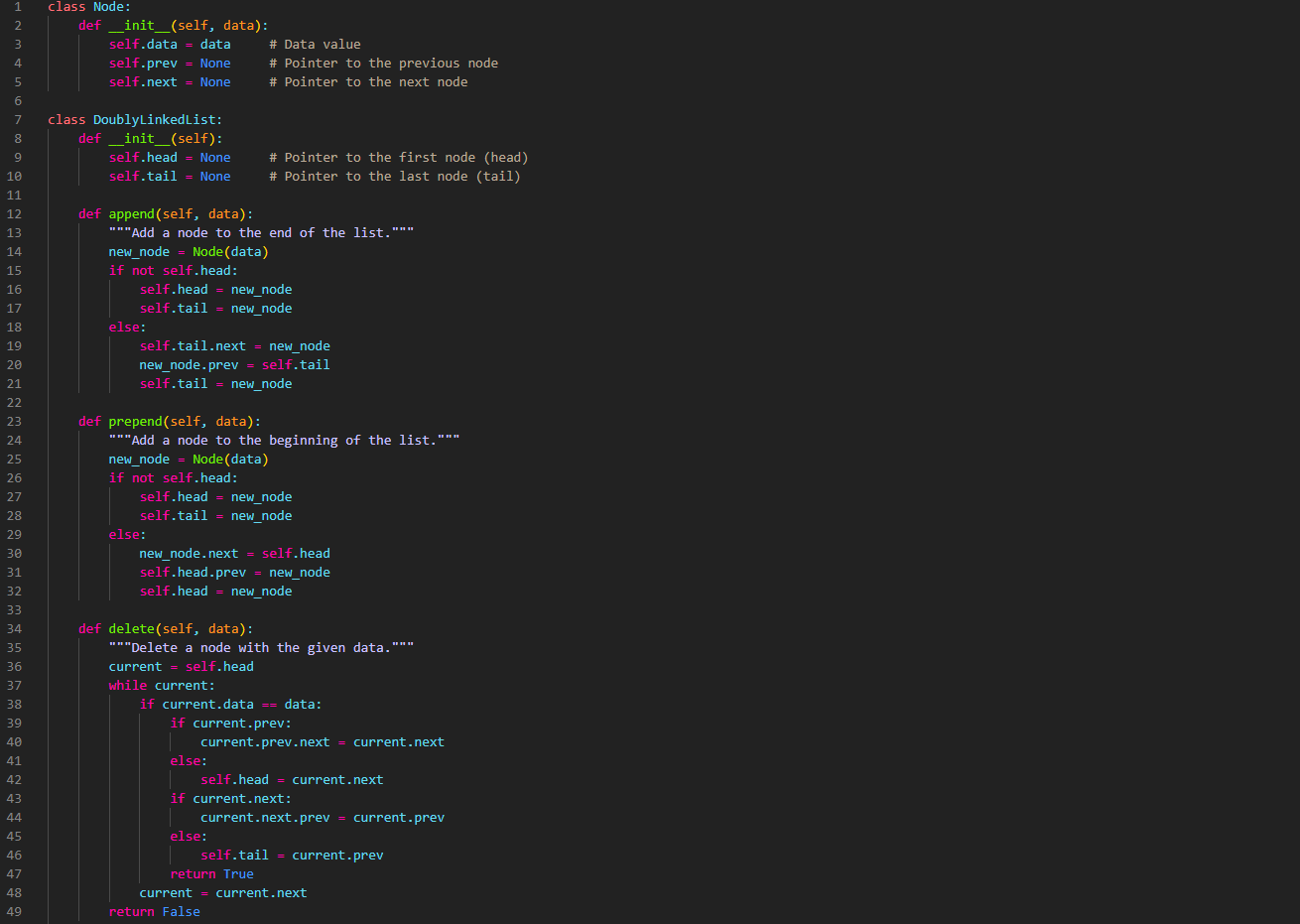
* **Prev Pointer** enables backward traversal, while **Next Pointer** allows forward traversal.

**Diagram Representation**

We’ll include a diagram to show the layout of nodes in a doubly linked list, including the head (first node) and tail (last node) pointers.

**4. Class Definitions**

Here’s a Python implementation for a doubly linked list, defining essential operations such as inserting, deleting, and traversing nodes:



A screen shot of a computer program

Description automatically generated

**5. Explanation of Core Operations**

1. **Append**: Adds a new node at the end of the list.
2. **Prepend**: Adds a new node at the beginning of the list.
3. **Delete**: Finds and removes the first node with the specified data.
4. **Display Forward**: Traverses the list from head to tail, displaying each node’s data.
5. **Display Backward**: Traverses the list from tail to head, displaying each node’s data.

**6. Complexity Analysis**

* **Insertion (at ends)**: O(1)O(1)O(1) — Direct access to the head and tail pointers enables efficient insertions at the ends.
* **Deletion (arbitrary node)**: O(n)O(n)O(n) — Traversing the list to find the node takes O(n)O(n)O(n), but deletion itself is O(1)O(1)O(1).
* **Traversal**: O(n)O(n)O(n) — Each node is visited once.

**7. Example Usage**

A computer screen shot of a black screen

Description automatically generated

**7. Diagram**

**A black and white image of a circle and arrows

Description automatically generated**