Doubly Linked Lists (DLL)

After singly linked lists, we've come to the more evolved version of the linked list data structure: doubly linked lists.

We'll cover the following

- Introduction
- Structure of the Doubly Linked List (DLL)
  - Impact on Deletion

Introduction

By now, you must have noticed a constraint which arises when dealing with singly linked lists. For any function which does not operate at the **head** node, we must traverse the whole list in a loop.

While the search operation in a normal list works in the same way, access is much faster as lists allow indexing.

Furthermore, since a linked list can only be traversed in one direction, we needlessly have to keep track of previous elements.

This is where the doubly linked list comes to the rescue!

Structure of the Doubly Linked List (DLL)

The only difference between doubly and singly linked lists is that in DLLs each node contains pointers for both the previous and the next node. This makes the DLLs **bi-directional**.

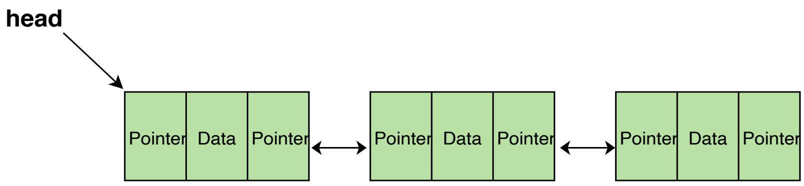
To implement this in code, we simply need to add a new member to the already constructed **Node** class:

```
class Node:
    def __init__(self, value):
        self.data = value  # Stores data
        self.previous_element = None  # Stores pointer to previous element
        self.next_element = None  # Stores pointer to next element

```

Explanation: `data` and `next_element` remain unchanged. The `previous_element` pointer has been introduced to store information about the preceding node.

Take a look at what the doubly linked list looks like:



Impact on Deletion

The addition of a backwards pointer significantly improves the searching process during deletion as you don't need to keep track of the previous node.

Let's rewrite the `delete` method from the previous lesson:

main.py

LinkedList.py

Node.py

```
from LinkedList import LinkedList
from Node import Node


def delete(lst, value):
    deleted = False
    if lst.is_empty():
        print("List is Empty")
        return deleted

    current_node = lst.get_head()

    if current_node.data is value:
        # Point head to the next element of the first element
        lst.head_node = current_node.next_element
        # Point the next element of the first element to Nobe
        current_node.next_element.previous_element = None
        deleted = True  # Both links have been changed.
        print(str(current_node.data) + " Deleted!")
        return deleted

    # Traversing/Searching for node to Delete
    while current_node:
        if value is current_node.data:
            if current_node.next_element:
                # Link the next node and the previous node to
                prev_node = current_node.previous_element
                next_node = current_node.next_element
                prev_node.next_element = next_node
                next_node.previous_element = prev_node
                # previous node pointer was maintained in Sing

            else:
                current_node.previous_element.next_element = N
            deleted = True
            break
        # previousNode = tempNode was used in Singly Linked Li
        current_node = current_node.next_element

    if deleted is False:
        print(str(value) + " is not in the List!")
    else:
        print(str(value) + " Deleted!")
    return deleted


lst = LinkedList()
```

```
48 for i in range(11):
49     lst.insert_at_head(i)
50
51 lst.print_list()
52 delete(lst, 5)
53
54 lst.print_list()
55 delete(lst, 0)
56
57 lst.print_list()
58
```

Most of the code is identical to the singly linked list implementation for deletion. However, we do not need to keep track of the previous node in the list.

Another difference is that on insertion and deletion we need to change two pointers rather than one.

For example, we cannot call the previously implemented `delete_at_head` function because deletion requires two steps now:

```
list.head_node = list.head_node.next_element
list.head_node.next_element.previous_element = None
```

The first line looks familiar from last time, but, in the second line, we specify the new backward link to `None` as well.

This principle holds for deletion anywhere in the list. **Line 17** follows it as well.

The one exception to this rule would be **deletion at the tail** because the last element only points to `None`.

By now, we can understand the logic behind doubly linked lists. Next, we'll see how they compare to singly linked lists in terms of performance and convenience.