**A doubly linked list is a data structure made up of nodes, where each node contains three parts:**

1. A data field to store data.

2. A reference to the next node in the sequence.

3. A reference to the previous node in the sequence.

This design allows traversal in both directions (forward and backward), making doubly linked lists more flexible than singly linked lists, where traversal is only possible in one direction.

**Here's a basic Python program to create and manipulate a doubly linked list:**

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.next = None

self.prev = None

class DoublyLinkedList:

def \_\_init\_\_(self):

self.head = None

**# Method to add a node at the end of the list**

def append(self, data):

new\_node = Node(data)

if self.head is None:

self.head = new\_node

return

last = self.head

while last.next:

last = last.next

last.next = new\_node

new\_node.prev = last

**# Method to display the list in forward direction**

def display\_forward(self):

current = self.head

while current:

print(current.data, end=" ")

current = current.next

print()

**# Method to display the list in reverse direction**

def display\_backward(self):

current = self.head

if current is None:

return

# Move to the last node

while current.next:

current = current.next

# Traverse backward

while current:

print(current.data, end=" ")

current = current.prev

print()

**# Usage example**

dll = DoublyLinkedList()

dll.append(1)

dll.append(2)

dll.append(3)

print("Doubly linked list in forward direction:")

dll.display\_forward()

print("Doubly linked list in backward direction:")

dll.display\_backward()

**### Explanation:**

1. Node Class: This defines a node with `data`, `next`, and `prev` attributes.

2. DoublyLinkedList Class: This contains methods to append nodes and display the list both forward and backward.

3. append() Method: Adds a new node at the end.

4. display\_forward() and display\_backward() Methods: Print the list in forward and backward directions respectively.

**### Output:**

Doubly linked list in forward direction:

1 2 3

Doubly linked list in backward direction:

3 2 1

This simple implementation allows you to create and manipulate a doubly linked list in Python.

Here’s a Python example of a doubly linked list that includes methods for insertion, deletion, traversing, searching, and checking if the list is empty. I'll also describe how each operation works.

**### Doubly Linked List Example with Basic Operations**

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data # Store data

self.next = None # Pointer to next node

self.prev = None # Pointer to previous node

class DoublyLinkedList:

def \_\_init\_\_(self):

self.head = None # Initialize the head as None

**# Check if the list is empty**

def is\_empty(self):

return self.head is None

**# Insert a node at the beginning**

def insert\_at\_beginning(self, data):

new\_node = Node(data)

if self.is\_empty(): # If list is empty, set head to new node

self.head = new\_node

else:

new\_node.next = self.head

self.head.prev = new\_node

self.head = new\_node

**# Insert a node at the end**

def insert\_at\_end(self, data):

new\_node = Node(data)

if self.is\_empty():

self.head = new\_node

else:

last = self.head

while last.next: # Traverse to the end of the list

last = last.next

last.next = new\_node

new\_node.prev = last

**# Delete a node by value**

def delete(self, key):

if self.is\_empty():

print("List is empty")

return

current = self.head

# Find the node with the given data

while current and current.data != key:

current = current.next

**# If node with the key is not found**

if current is None:

print(f"Node with data {key} not found.")

return

**# If the node is the head node**

if current == self.head:

self.head = current.next

if self.head:

self.head.prev = None

else:

if current.next: # If node to delete is not the last node

current.next.prev = current.prev

if current.prev:

current.prev.next = current.next

**# Search for a node by value**

def search(self, key):

current = self.head

while current:

if current.data == key:

return True # Return True if data is found

current = current.next

return False # Return False if not found

**# Traverse the list in forward direction**

def display\_forward(self):

current = self.head

while current:

print(current.data, end=" <-> " if current.next else "")

current = current.next

print()

**# Traverse the list in backward direction**

def display\_backward(self):

current = self.head

if current is None:

return

**# Move to the end**

while current.next:

current = current.next

**# Traverse backward**

while current:

print(current.data, end=" <-> " if current.prev else "")

current = current.prev

print()

**# Usage example**

dll = DoublyLinkedList()

print("Inserting nodes at the beginning:")

dll.insert\_at\_beginning(10)

dll.insert\_at\_beginning(20)

dll.display\_forward() # Expected output: 20 <-> 10

print("\nInserting nodes at the end:")

dll.insert\_at\_end(30)

dll.insert\_at\_end(40)

dll.display\_forward() # Expected output: 20 <-> 10 <-> 30 <-> 40

print("\nDeleting a node (value: 10):")

dll.delete(10)

dll.display\_forward() # Expected output: 20 <-> 30 <-> 40

print("\nSearching for a node (value: 30):")

print("Found" if dll.search(30) else "Not found") # Expected output: Found

print("\nChecking if the list is empty:")

print("List is empty" if dll.is\_empty() else "List is not empty") # Expected output: List is not empty

print("\nDisplaying list in reverse:")

dll.display\_backward() # Expected output: 40 <-> 30 <-> 20

**### Explanation of Each Operation**

1. Insertion:

- At the Beginning: Adds a new node before the head. Adjusts the new node’s `next` to point to the old head, and the old head’s `prev` to the new node.

- At the End: Adds a new node after the last node. Sets the last node’s `next` to the new node and the new node’s `prev` to the last node.

2. Deletion:

- Deletes a node with the specified value. Adjusts pointers of neighboring nodes to bypass the node being deleted.

3. Traversing:

- Forward: Starts at the head and moves through `next` pointers.

- Backward: Moves to the last node by following `next` pointers, then traverses backward using `prev` pointers.

4. Searching:

- Finds if a node with a specific value exists by checking each node’s data field.

5. Checking for Empty List:

- Returns `True` if the head is `None`, indicating the list is empty.

**### Visual Representation of Operations**

1. Insertion at Beginning:

- List before: `None`

- Insert `10`: `10 <-> None`

- Insert `20` at the beginning: `20 <-> 10`

2. Insertion at End:

- List before: `20 <-> 10`

- Insert `30` at end: `20 <-> 10 <-> 30`

- Insert `40` at end: `20 <-> 10 <-> 30 <-> 40`

3. Deletion:

- List before: `20 <-> 10 <-> 30 <-> 40`

- Delete `10`: `20 <-> 30 <-> 40`

4. Traversal Forward and Backward:

- Forward from head: `20 <-> 30 <-> 40`

- Backward from end: `40 <-> 30 <-> 20`

This structure provides a clear example of a doubly linked list with basic operations in Python.

**QUESTIONS:**

1. Define the `Node` Class

Create a `Node` class to represent each node in the doubly linked list. This class should have the following attributes:

- `data`: Holds the value of the node.

- `next`: Points to the next node in the list.

- `prev`: Points to the previous node in the list.

2. Define the `DLL` Class

Create a `DLL` (Doubly Linked List) class to implement the operations of a doubly linked list. This class should contain an `\_\_init\_\_()` method to initialize the `start` attribute, which represents the head of the list and is initially set to `None`.

3. Define the `is\_empty()` Method

In the `DLL` class, define the `is\_empty()` method to check if the linked list is empty. This method should return `True` if the list is empty (i.e., `start` is `None`), and `False` otherwise.

4. Define the `insert\_at\_start()` Method

In the `DLL` class, define the `insert\_at\_start()` method to insert a new node at the beginning of the list. This method should:

- Create a new `Node` with the specified data.

- If the list is empty, set `start` to this new node.

- Otherwise, update the `next` pointer of the new node to the current `start`, update the `prev` pointer of the current `start` to the new node, and set `start` to the new node.

5. Define the `insert\_at\_last()` Method

In the `DLL` class, define the `insert\_at\_last()` method to insert a new node at the end of the list. This method should:

- Create a new `Node` with the specified data.

- If the list is empty, set `start` to this new node.

- Otherwise, traverse to the end of the list, update the `next` pointer of the last node to the new node, and set the `prev` pointer of the new node to the last node.

6. Define the `search()` Method

In the `DLL` class, define the `search()` method to find and return the node with the specified data value. This method should:

- Traverse the list from the start, checking each node’s `data`.

- Return the node if found, or `None` if the value is not in the list.

7. Define the `insert\_after()` Method

In the `DLL` class, define the `insert\_after()` method to insert a new node after a node containing a specified value. This method should:

- Search for the node with the specified value using the `search()` method.

- If found, create a new node with the given data.

- Update the pointers of the relevant nodes to insert the new node after the specified node.

8. Define the `display()` Method

In the `DLL` class, define the `display()` method to print all elements in the list from start to end. This method should:

- Traverse the list starting from `start`, printing each node's data until reaching the end of the list.

9. Implement an Iterator for `DLL`

In the `DLL` class, implement an iterator to enable easy access to all elements in the list using a `for` loop. This requires:

- Defining an `\_\_iter\_\_()` method that sets up iteration by initializing an internal pointer to `start`.

- Defining a `\_\_next\_\_()` method to return the next node’s data and advance the pointer, raising `StopIteration` at the end of the list.

**Program:**

```python

class Node:

def \_\_init\_\_(self, data):

self.data = data # Data of the node

self.next = None # Pointer to the next node

self.prev = None # Pointer to the previous node

class DLL:

def \_\_init\_\_(self):

self.start = None # Initialize the start of the list as None

**# Method to check if the list is empty**

def is\_empty(self):

return self.start is None

**# Method to insert an element at the start of the list**

def insert\_at\_start(self, data):

new\_node = Node(data)

if self.is\_empty():

self.start = new\_node

else:

new\_node.next = self.start

self.start.prev = new\_node

self.start = new\_node

**# Method to insert an element at the end of the list**

def insert\_at\_last(self, data):

new\_node = Node(data)

if self.is\_empty():

self.start = new\_node

else:

last = self.start

while last.next: # Traverse to the end of the list

last = last.next

last.next = new\_node

new\_node.prev = last

**# Method to search for a node with a specific value**

def search(self, key):

current = self.start

while current:

if current.data == key:

return current # Return the node if found

current = current.next

return None # Return None if not found

**# Method to insert a new node after a given node with a specific value**

def insert\_after(self, key, data):

current = self.search(key)

if current is None:

print(f"Node with data {key} not found.")

return

new\_node = Node(data)

new\_node.next = current.next

new\_node.prev = current

if current.next:

current.next.prev = new\_node

current.next = new\_node

**# Method to print all elements of the list**

def display(self):

current = self.start

while current:

print(current.data, end=" <-> " if current.next else "")

current = current.next

print()

**# Iterator method to access all elements of the list**

def \_\_iter\_\_(self):

self.\_iter\_node = self.start

return self

def \_\_next\_\_(self):

if self.\_iter\_node is None:

raise StopIteration

data = self.\_iter\_node.data

self.\_iter\_node = self.\_iter\_node.next

return data

**# Usage example**

dll = DLL()

**# Inserting elements at the start**

dll.insert\_at\_start(10)

dll.insert\_at\_start(20)

**# Inserting elements at the end**

dll.insert\_at\_last(30)

dll.insert\_at\_last(40)

**# Displaying the list**

print("List after insertions:")

dll.display() # Expected output: 20 <-> 10 <-> 30 <-> 40

**# Searching for an element**

print("Searching for 30:")

found\_node = dll.search(30)

print("Found" if found\_node else "Not found") # Expected output: Found

**# Inserting after a specific element**

print("Inserting 25 after 20:")

dll.insert\_after(20, 25)

dll.display() # Expected output: 20 <-> 25 <-> 10 <-> 30 <-> 40

**# Checking if list is empty**

print("Checking if list is empty:")

print("List is empty" if dll.is\_empty() else "List is not empty") # Expected output: List is not empty

**# Using iterator to access all elements**

print("List elements using iterator:")

for value in dll:

print(value, end=" <-> " if value != 40 else "")

print()

**### Explanation**

1. Node Class:

- Represents each node in the list, with `data`, `next`, and `prev` attributes.

2. DLL Class:

- `\_\_init\_\_`: Initializes the doubly linked list with `start` as `None`.

- `is\_empty`: Checks if the list is empty by confirming if `start` is `None`.

- `insert\_at\_start`: Inserts a new node at the beginning of the list.

- `insert\_at\_last`: Inserts a new node at the end of the list.

- `search`: Searches for a node by value and returns the node if found.

- `insert\_after`: Inserts a new node after the node with the specified key.

- `display`: Prints all elements in the list.

- `\_\_iter\_\_` and `\_\_next\_\_`: Make the `DLL` class iterable, allowing us to loop through each node's data in the list.

### Example Output

List after insertions:

20 <-> 10 <-> 30 <-> 40

Searching for 30:

Found

Inserting 25 after 20:

20 <-> 25 <-> 10 <-> 30 <-> 40

Checking if list is empty:

List is not empty

List elements using iterator:

20 <-> 25 <-> 10 <-> 30 <-> 40

This example provides a complete implementation of a doubly linked list with all required operations, including an iterator to easily access each element in the list.