A deque (double-ended queue) is a data structure that allows insertion and deletion of elements from both ends. Unlike a regular queue, where you can only enqueue at the rear and dequeue from the front, a deque allows operations at both the front and rear. This provides greater flexibility and makes deques suitable for scenarios where elements need to be processed from both ends.

### Differences Between Queue and Deque

- Queue: Allows insertion at the rear and deletion from the front only.

- Deque: Allows insertion and deletion at both the front and rear.

### Operations on Deque

Here are the main operations on a deque, with examples:

1. Insert Front: Adds an element to the front.

```python

deque.insert\_front(10)

```

2. Insert Rear: Adds an element to the rear.

```python

deque.insert\_rear(20)

```

3. Delete Front: Removes the element from the front.

```python

deque.delete\_front()

```

4. Delete Rear: Removes the element from the rear.

```python

deque.delete\_rear()

```

5. Get Front: Retrieves the element from the front without removing it.

```python

deque.get\_front()

```

6. Get Rear: Retrieves the element from the rear without removing it.

```python

deque.get\_rear()

```

7. Is Empty: Checks if the deque is empty.

```python

deque.is\_empty()

```

8. Size: Returns the number of elements in the deque.

```python

deque.size()

```

### Implementations of Deque

Here are several ways to implement a deque:

#### 1. Using List

```python

class Deque:

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

self.deque = []

def insert\_front(self, item):

self.deque.insert(0, item)

def insert\_rear(self, item):

self.deque.append(item)

def delete\_front(self):

if self.is\_empty():

raise IndexError("Deque is empty")

return self.deque.pop(0)

def delete\_rear(self):

if self.is\_empty():

raise IndexError("Deque is empty")

return self.deque.pop()

def get\_front(self):

return self.deque[0] if not self.is\_empty() else None

def get\_rear(self):

return self.deque[-1] if not self.is\_empty() else None

def is\_empty(self):

return len(self.deque) == 0

def size(self):

return len(self.deque)

```

#### 2. By Extending List Class

```python

class DequeList(list):

def insert\_front(self, item):

self.insert(0, item)

def insert\_rear(self, item):

self.append(item)

def delete\_front(self):

return self.pop(0)

def delete\_rear(self):

return self.pop()

def get\_front(self):

return self[0]

def get\_rear(self):

return self[-1]

def is\_empty(self):

return len(self) == 0

def size(self):

return len(self)

```

#### 3. Using Doubly Linked List

```python

class Node:

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

self.data = data

self.next = None

self.prev = None

class DoublyLinkedDeque:

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

self.front = None

self.rear = None

self.\_size = 0

def insert\_front(self, item):

new\_node = Node(item)

if self.is\_empty():

self.front = self.rear = new\_node

else:

new\_node.next = self.front

self.front.prev = new\_node

self.front = new\_node

self.\_size += 1

def insert\_rear(self, item):

new\_node = Node(item)

if self.is\_empty():

self.front = self.rear = new\_node

else:

new\_node.prev = self.rear

self.rear.next = new\_node

self.rear = new\_node

self.\_size += 1

def delete\_front(self):

if self.is\_empty():

raise IndexError("Deque is empty")

data = self.front.data

self.front = self.front.next

if self.front:

self.front.prev = None

else:

self.rear = None

self.\_size -= 1

return data

def delete\_rear(self):

if self.is\_empty():

raise IndexError("Deque is empty")

data = self.rear.data

self.rear = self.rear.prev

if self.rear:

self.rear.next = None

else:

self.front = None

self.\_size -= 1

return data

def get\_front(self):

return self.front.data if self.front else None

def get\_rear(self):

return self.rear.data if self.rear else None

def is\_empty(self):

return self.front is None

def size(self):

return self.\_size

```

#### 4. By Extending Doubly Linked List Class

Extending a class-based doubly linked list with deque operations is possible but requires you to add specialized methods for front and rear insertion and deletion.

#### 5. Using Linked List Concept

For a deque, a doubly linked list is preferred over a singly linked list because both ends need to be accessible without extensive traversal. A singly linked list does not allow efficient access to both the front and rear due to its one-way link structure.

### Why Not Use Singly Linked List?

A singly linked list would not be ideal for implementing a deque because:

- Accessing the rear of a singly linked list is inefficient, requiring traversal from the head to the end for every rear operation.

- A doubly linked list provides direct access to both ends, enabling efficient insertion and deletion from both front and rear

Here is an implementation of a `Deque` class using a Python list, as per the requirements:

```python

class Deque:

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

# Step 1: Initialize an empty list for the deque

self.deque = []

# Step 2: Method to check if the deque is empty

def is\_empty(self):

return len(self.deque) == 0

# Step 3: Method to add data at the front of the deque

def insert\_front(self, data):

self.deque.insert(0, data)

# Step 4: Method to add data at the rear of the deque

def insert\_rear(self, data):

self.deque.append(data)

# Step 5: Method to remove the front element from the deque

def delete\_front(self):

if self.is\_empty():

raise IndexError("Deque is empty, cannot delete front element")

return self.deque.pop(0)

# Step 6: Method to remove the rear element from the deque

def delete\_rear(self):

if self.is\_empty():

raise IndexError("Deque is empty, cannot delete rear element")

return self.deque.pop()

# Step 7: Method to get the front element of the deque

def get\_front(self):

if self.is\_empty():

return None

return self.deque[0]

# Step 8: Method to get the rear element of the deque

def get\_rear(self):

if self.is\_empty():

return None

return self.deque[-1]

# Step 9: Method to get the size of the deque

def size(self):

return len(self.deque)

```

### Example Usage

```python

# Creating an instance of Deque

dq = Deque()

# Checking if the deque is empty

print("Is deque empty?", dq.is\_empty())

# Inserting elements at the front and rear

dq.insert\_front(10)

dq.insert\_rear(20)

dq.insert\_front(5)

print("Deque after insertions:", dq.deque)

# Getting the front and rear elements

print("Front element:", dq.get\_front())

print("Rear element:", dq.get\_rear())

# Checking the size of the deque

print("Size of deque:", dq.size())

# Deleting elements from the front and rear

dq.delete\_front()

dq.delete\_rear()

print("Deque after deletions:", dq.deque)

# Final check for size and empty status

print("Size of deque:", dq.size())

print("Is deque empty?", dq.is\_empty())

```

### Explanation of Methods

1. `\_\_init\_\_`: Initializes an empty list for deque storage.

2. `is\_empty`: Returns `True` if deque is empty; `False` otherwise.

3. `insert\_front`: Adds an element to the front using `insert(0, data)`.

4. `insert\_rear`: Adds an element to the rear using `append(data)`.

5. `delete\_front`: Removes and returns the front element, raises an error if deque is empty.

6. `delete\_rear`: Removes and returns the rear element, raises an error if deque is empty.

7. `get\_front`: Returns the front element without removing it.

8. `get\_rear`: Returns the rear element without removing it.

9. `size`: Returns the number of elements in the deque.

This provides a complete and functional deque implementation using a list.

Here's the corrected and properly formatted implementation of the `Deque` class, based on your provided code. The code has been refined to ensure correct handling of the `is\_empty` method, and some minor logic improvements have been made:

```python

from dataclasses import dataclass

class Deque:

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

# Initialize an empty list to hold deque items

self.items = []

def is\_empty(self):

# Check if the deque is empty

return len(self.items) == 0

def insert\_front(self, data):

# Add data at the front of the deque

self.items.insert(0, data)

def insert\_rear(self, data):

# Add data at the rear of the deque

self.items.append(data)

def delete\_front(self):

# Remove and return the front element, raise error if deque is empty

if self.is\_empty():

raise IndexError("Deque is empty")

return self.items.pop(0)

def delete\_rear(self):

# Remove and return the rear element, raise error if deque is empty

if self.is\_empty():

raise IndexError("Deque is empty")

return self.items.pop()

def get\_front(self):

# Return the front element without removing it, raise error if deque is empty

if self.is\_empty():

raise IndexError("Deque is empty")

return self.items[0]

def get\_rear(self):

# Return the rear element without removing it, raise error if deque is empty

if self.is\_empty():

raise IndexError("Deque is empty")

return self.items[-1]

def size(self):

# Return the number of items in the deque

return len(self.items)

# Example usage

d1 = Deque()

d1.insert\_front(10)

d1.insert\_front(20)

d1.insert\_rear(30)

d1.insert\_rear(40)

print("Front element:", d1.get\_front()) # Output: 20

print("Rear element:", d1.get\_rear()) # Output: 40

print("Size of deque:", d1.size()) # Output: 4

```

### Explanation of Improvements

1. Correct `is\_empty` Method: This method now correctly checks if the deque is empty by comparing the length of the `items` list to zero.

2. Proper Error Handling: The `get\_front`, `get\_rear`, `delete\_front`, and `delete\_rear` methods now raise an `IndexError` if the deque is empty.

3. Example Usage: An example of using the `Deque` class has been provided for clarity, showing how to insert and retrieve elements.

This corrected version ensures that all operations on the deque are handled properly and are consistent with the deque's intended functionality.

Here is the implementation of a `Deque` using a doubly linked list:

```python

class Node:

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

# Step 1: Define Node with prev, item, and next attributes

self.item = item

self.prev = None

self.next = None

class Deque:

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

# Step 2: Initialize the front and rear of the deque and an item count

self.front = None

self.rear = None

self.item\_count = 0

# Step 3: Method to check if the deque is empty

def is\_empty(self):

return self.item\_count == 0

# Step 4: Method to add an item at the front

def insert\_front(self, item):

new\_node = Node(item)

if self.is\_empty():

self.front = self.rear = new\_node

else:

new\_node.next = self.front

self.front.prev = new\_node

self.front = new\_node

self.item\_count += 1

# Step 5: Method to add an item at the rear

def insert\_rear(self, item):

new\_node = Node(item)

if self.is\_empty():

self.front = self.rear = new\_node

else:

new\_node.prev = self.rear

self.rear.next = new\_node

self.rear = new\_node

self.item\_count += 1

# Step 6: Method to remove the front item

def delete\_front(self):

if self.is\_empty():

raise IndexError("Deque is empty")

removed\_item = self.front.item

self.front = self.front.next

if self.front:

self.front.prev = None

else:

self.rear = None

self.item\_count -= 1

return removed\_item

# Step 7: Method to remove the rear item

def delete\_rear(self):

if self.is\_empty():

raise IndexError("Deque is empty")

removed\_item = self.rear.item

self.rear = self.rear.prev

if self.rear:

self.rear.next = None

else:

self.front = None

self.item\_count -= 1

return removed\_item

# Step 8: Method to get the front item

def get\_front(self):

if self.is\_empty():

raise IndexError("Deque is empty")

return self.front.item

# Step 9: Method to get the rear item

def get\_rear(self):

if self.is\_empty():

raise IndexError("Deque is empty")

return self.rear.item

# Step 10: Method to get the size of the deque

def size(self):

return self.item\_count

# Example usage

dq = Deque()

dq.insert\_front(10)

dq.insert\_rear(20)

dq.insert\_front(5)

dq.insert\_rear(30)

print("Front element:", dq.get\_front()) # Output: 5

print("Rear element:", dq.get\_rear()) # Output: 30

print("Size of deque:", dq.size()) # Output: 4

dq.delete\_front() # Removes 5

dq.delete\_rear() # Removes 30

print("Front element after deletion:", dq.get\_front()) # Output: 10

print("Rear element after deletion:", dq.get\_rear()) # Output: 20

print("Size of deque after deletion:", dq.size()) # Output: 2

```

### Explanation of Each Method

1. `Node` class: Represents each node with `prev`, `item`, and `next`.

2. `Deque` class `\_\_init\_\_`: Initializes `front`, `rear`, and `item\_count`.

3. `is\_empty`: Checks if the deque is empty based on `item\_count`.

4. `insert\_front`: Inserts a node at the front; adjusts `front` and `rear` if necessary.

5. `insert\_rear`: Inserts a node at the rear; adjusts `front` and `rear` if necessary.

6. `delete\_front`: Removes the front node, handling cases where deque becomes empty.

7. `delete\_rear`: Removes the rear node, handling cases where deque becomes empty.

8. `get\_front`: Returns the front item without removal.

9. `get\_rear`: Returns the rear item without removal.

10. `size`: Returns the current item count.

This provides a complete doubly linked list-based `Deque` class with all required functionalities.

Here is the corrected and properly formatted version of your `Deque` class with comments and improved readability:

```python

class Node:

def \_\_init\_\_(self, item=None, prev=None, next=None):

# Initialize a node with item, prev, and next pointers

self.item = item

self.prev = prev

self.next = next

class Deque:

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

# Initialize an empty deque with front, rear pointers and an item count

self.front = None

self.rear = None

self.item\_count = 0

def is\_empty(self):

# Check if deque is empty

return self.item\_count == 0

def insert\_front(self, data):

# Insert a new node at the front

new\_node = Node(data, None, self.front)

if self.is\_empty():

self.rear = new\_node # If deque was empty, rear is also the new node

else:

self.front.prev = new\_node

self.front = new\_node

self.item\_count += 1

def insert\_rear(self, data):

# Insert a new node at the rear

new\_node = Node(data, self.rear, None)

if self.is\_empty():

self.front = new\_node # If deque was empty, front is also the new node

else:

self.rear.next = new\_node

self.rear = new\_node

self.item\_count += 1

def delete\_front(self):

# Remove the front node

if self.is\_empty():

raise IndexError("Deque is empty")

if self.front == self.rear:

# If there is only one node

self.front = None

self.rear = None

else:

self.front = self.front.next

self.front.prev = None

self.item\_count -= 1

def delete\_rear(self):

# Remove the rear node

if self.is\_empty():

raise IndexError("Deque is empty")

if self.front == self.rear:

# If there is only one node

self.front = None

self.rear = None

else:

self.rear = self.rear.prev

self.rear.next = None

self.item\_count -= 1

def get\_front(self):

# Return the front item

if self.is\_empty():

raise IndexError("Deque is empty")

return self.front.item

def get\_rear(self):

# Return the rear item

if self.is\_empty():

raise IndexError("Deque is empty")

return self.rear.item

def size(self):

# Return the number of items in the deque

return self.item\_count

# Example usage

dq = Deque()

dq.insert\_front(10)

dq.insert\_rear(20)

dq.insert\_front(5)

dq.insert\_rear(30)

print("Front element:", dq.get\_front()) # Output: 5

print("Rear element:", dq.get\_rear()) # Output: 30

print("Size of deque:", dq.size()) # Output: 4

dq.delete\_front() # Removes 5

dq.delete\_rear() # Removes 30

print("Front element after deletion:", dq.get\_front()) # Output: 10

print("Rear element after deletion:", dq.get\_rear()) # Output: 20

print("Size of deque after deletion:", dq.size()) # Output: 2

```

### Explanation of Corrections

1. Indentation: Corrected indentation errors.

2. Increment/Decrement of `item\_count`: Corrected `item\_count` adjustments in `delete\_front` and `delete\_rear` methods.

3. Error Handling: Replaced `return IndexError(...)` with `raise IndexError(...)` for exceptions.

4. Proper Linking: Corrected the handling of `prev` and `next` pointers in insertions and deletions.

5. Consistency: Added `item\_count` adjustments in `insert\_front` and `insert\_rear` methods.

This corrected version provides a fully functional `Deque` with doubly linked list implementation.