### Queue Data Structure and Its Working Principle

A queue is a linear data structure that follows the FIFO (First-In-First-Out) principle. This means that elements are inserted at the rear (end) of the queue and removed from the front (beginning).

#### Why Queue is an Abstract Data Type (ADT)

A queue is considered an ADT because it defines a specific behavior (FIFO), regardless of how it is implemented. Users of a queue interact with it through its interface (enqueue, dequeue, etc.) without needing to know its internal implementation.

#### Front and Rear in a Queue

- Rear: The end of the queue where new elements are added.

- Front: The beginning of the queue where elements are removed.

#### Operations of Queue

1. enqueue(data): Adds an element at the rear of the queue.

2. dequeue(): Removes the front element from the queue.

3. is\_empty(): Checks if the queue is empty.

4. get\_front(): Returns the front element of the queue.

5. get\_rear(): Returns the rear element of the queue.

6. size(): Returns the number of elements in the queue.

#### Real-World Examples

- Line at a ticket counter: The first person in line is the first to be served.

- Printer queue: Documents are printed in the order they were added to the queue.

---

### Programming Examples and Implementation of Queue

#### Implementation 1: Queue Using a List

```python

class Queue:

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

# Initialize an empty list to store the queue elements

self.items = []

def is\_empty(self):

# Check if the queue is empty

return len(self.items) == 0

def enqueue(self, data):

# Add an element to the rear of the queue

self.items.append(data)

def dequeue(self):

# Remove an element from the front of the queue if it's not empty

if not self.is\_empty():

return self.items.pop(0)

else:

raise IndexError("Queue is empty")

def get\_front(self):

# Return the front element of the queue

if not self.is\_empty():

return self.items[0]

else:

raise IndexError("Queue is empty")

def get\_rear(self):

# Return the rear element of the queue

if not self.is\_empty():

return self.items[-1]

else:

raise IndexError("Queue is empty")

def size(self):

# Return the number of elements in the queue

return len(self.items)

```

#### Implementation 2: Queue by Extending List Class

```python

class Queue(list):

def is\_empty(self):

return len(self) == 0

def enqueue(self, data):

self.append(data)

def dequeue(self):

if not self.is\_empty():

return self.pop(0)

else:

raise IndexError("Queue is empty")

def get\_front(self):

if not self.is\_empty():

return self[0]

else:

raise IndexError("Queue is empty")

def get\_rear(self):

if not self.is\_empty():

return self[-1]

else:

raise IndexError("Queue is empty")

def size(self):

return len(self)

```

#### Implementation 3: Queue Using a Singly Linked List

```python

class Node:

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

self.data = data

self.next = None

class Queue:

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

self.front = None

self.rear = None

self.item\_count = 0

def is\_empty(self):

return self.front is None

def enqueue(self, data):

new\_node = Node(data)

if self.is\_empty():

self.front = new\_node

self.rear = new\_node

else:

self.rear.next = new\_node

self.rear = new\_node

self.item\_count += 1

def dequeue(self):

if self.is\_empty():

raise IndexError("Queue is empty")

removed\_data = self.front.data

self.front = self.front.next

if self.front is None:

self.rear = None

self.item\_count -= 1

return removed\_data

def get\_front(self):

if self.is\_empty():

raise IndexError("Queue is empty")

return self.front.data

def get\_rear(self):

if self.is\_empty():

raise IndexError("Queue is empty")

return self.rear.data

def size(self):

return self.item\_count

```

---

### Assignments

#### Assignment-12: Queue Using List

1. Define a class `Queue`: Implement the queue using a list.

2. Define `\_\_init\_\_` method: Create an empty list as the instance attribute.

3. Define `is\_empty` method: Check if the list is empty.

4. Define `enqueue` method: Add data to the rear.

5. Define `dequeue` method: Remove data from the front.

6. Define `get\_front` method: Return the front element.

7. Define `get\_rear` method: Return the rear element.

8. Define `size` method: Return the number of items.

#### Assignment-13: Queue Using Singly Linked List Concept

1. Define a class `Queue`: Implement the queue with a singly linked list.

2. Define `\_\_init\_\_` method: Initialize front, rear, and item count.

3. Define `is\_empty` method: Check if the queue is empty.

4. Define `enqueue` method: Add data to the rear.

5. Define `dequeue` method: Remove the front element.

6. Define `get\_front` method: Return the front element.

7. Define `get\_rear` method: Return the rear element.

8. Define `size` method: Return the number of elements in the queue.

class Queue:  
 def \_\_init\_\_(self):  
 # Initialize an empty list to store queue elements  
 self.items = []  
 # Initialize front and rear to None, as the queue is empty initially  
 self.front = None  
 self.rear = None  
  
 def is\_empty(self):  
 # Check if the queue is empty by checking if items list has any elements  
 return len(self.items) == 0  
  
 def enqueue(self, data):  
 # Add a new element at the end of the queue (rear end)  
 self.items.append(data)  
 # Update rear pointer to the last element  
 self.rear = self.items[-1]  
 # Update front pointer to the first element, as it's still the first element after enqueue  
 self.front = self.items[0]  
  
 def dequeue(self):  
 # Remove an element from the front of the queue if it's not empty  
 if not self.is\_empty():  
 # Remove the first element from the items list  
 self.items.pop(0)  
 # Update front and rear pointers after dequeue  
 if len(self.items) > 0:  
 self.front = self.items[0] # Front points to new first item  
 self.rear = self.items[-1] # Rear points to last item  
 else:  
 # If queue is empty after dequeue, set front and rear to None  
 self.front = None  
 self.rear = None  
 else:  
 # If queue is empty, raise an error indicating underflow  
 raise IndexError("Queue underflow")  
  
 def get\_front(self):  
 # Return the front element if queue is not empty  
 if not self.is\_empty():  
 return self.items[0]  
 else:  
 # Raise error if the queue is empty  
 raise IndexError("Queue underflow")  
  
 def get\_rear(self):  
 # Return the rear element if queue is not empty  
 if not self.is\_empty():  
 return self.items[-1]  
 else:  
 # Raise error if the queue is empty  
 raise IndexError("Queue underflow")  
  
 def size(self):  
 # Return the number of elements in the queue  
 return len(self.items)  
  
  
# Initialize a Queue object  
Q1 = Queue()  
  
# Try to print the front element of the queue  
try:  
 print("Front element:", Q1.get\_front())  
except IndexError as e:  
 # If the queue is empty, an IndexError is raised, and we catch it here  
 print("Error:", e.args[0])  
  
# Add elements to the queue  
Q1.enqueue(10)  
Q1.enqueue(20)  
Q1.enqueue(30)  
Q1.enqueue(40)  
  
# Display the front and rear elements after enqueueing elements  
print("Front =", Q1.get\_front(), "Rear =", Q1.get\_rear())  
  
# Try to remove an element from the front of the queue  
try:  
 Q1.dequeue()  
 # Display the size of the queue after one element is dequeued  
 print("Queue has now", Q1.size(), "elements")  
except IndexError as e:  
 # If there is an underflow, catch the error and display a message  
 print("Error:", e.args[0])  
  
  
# ### Explanation of Key Parts  
# - \*\*Class Initialization (`\_\_init\_\_`)\*\*: Initializes an empty queue with `front` and `rear` pointers set to `None`.  
# - \*\*is\_empty\*\*: Checks if the queue has any elements by evaluating if `items` list is empty.  
# - \*\*enqueue\*\*: Adds an element to the end of the queue and updates `rear` and `front` pointers.  
# - \*\*dequeue\*\*: Removes the front element, updating `front` and `rear` pointers accordingly.  
# - \*\*get\_front\*\* and \*\*get\_rear\*\*: Retrieve the front and rear elements, respectively, with error handling for an empty queue.  
# - \*\*size\*\*: Returns the number of elements currently in the queue.

# Here's the code rewritten with detailed comments for each part to help you understand each operation and the queue implementation using a linked list.  
#  
# ```python  
# Define a Node class to represent each element (or node) in the linked list  
class Node:  
 def \_\_init\_\_(self, item=None, next=None):  
 # Initialize the node with the given item and the reference to the next node  
 self.item = item  
 self.next = next  
  
# Define a Queue class using a linked list structure  
class Queue:  
 def \_\_init\_\_(self):  
 # Initialize front and rear pointers to None, indicating an empty queue  
 self.front = None  
 self.rear = None  
 # Initialize item\_count to track the number of elements in the queue  
 self.item\_count = 0  
  
 def is\_empty(self):  
 # Check if the queue is empty by seeing if the front pointer is None  
 return self.front == None  
  
 def enqueue(self, data):  
 # Create a new node with the given data  
 n = Node(data)  
 if self.is\_empty():  
 # If the queue is empty, the new node becomes both front and rear  
 self.front = n  
 self.rear = n  
 else:  
 # Link the new node to the current rear node  
 self.rear.next = n  
 # Update the rear pointer to the new node  
 self.rear = n  
 # Increase the count of items in the queue  
 self.item\_count += 1  
  
 def dequeue(self):  
 # Remove an element from the front of the queue if it's not empty  
 if self.is\_empty():  
 # Raise an error if the queue is empty  
 raise IndexError("empty queue")  
 elif self.front == self.rear:  
 # If there's only one element, set both front and rear to None  
 self.front = None  
 self.rear = None  
 else:  
 # Move the front pointer to the next node in the queue  
 self.front = self.front.next  
 # Decrease the count of items in the queue  
 self.item\_count -= 1  
  
 def get\_front(self):  
 # Return the front element if the queue is not empty  
 if self.is\_empty():  
 raise IndexError("No data in the queue")  
 else:  
 return self.front.item  
  
 def get\_rear(self):  
 # Return the rear element if the queue is not empty  
 if self.is\_empty():  
 raise IndexError("No data in the queue")  
 else:  
 return self.rear.item  
  
 def size(self):  
 # Return the number of elements in the queue  
 return self.item\_count  
  
  
# Create a Queue object  
q1 = Queue()  
  
# Add elements to the queue  
q1.enqueue(10)  
q1.enqueue(20)  
q1.enqueue(30)  
q1.enqueue(40)  
  
# Display the front and rear elements of the queue  
print("Front:", q1.get\_front(), "Rear:", q1.get\_rear())  
  
# Remove an element from the front of the queue  
q1.dequeue()  
  
# Display the front and rear elements after dequeue  
print("Front:", q1.get\_front(), "Rear:", q1.get\_rear())  
  
### Explanation of Key Parts  
# - \*\*Node Class (`Node`)\*\*: Represents each element in the linked list with data (`item`) and a pointer to the next node (`next`).  
# - \*\*Queue Initialization (`\_\_init\_\_`)\*\*: Sets up an empty queue with `front`, `rear`, and `item\_count` initialized.  
# - \*\*is\_empty\*\*: Checks if the queue has any elements by verifying if `front` is `None`.  
# - \*\*enqueue\*\*: Adds a new node with `data` to the end of the queue, updating `rear` and linking nodes.  
# - \*\*dequeue\*\*: Removes the node at the `front`, updating pointers accordingly and raising an error if the queue is empty.  
# - \*\*get\_front\*\* and \*\*get\_rear\*\*: Retrieve the `front` and `rear` elements, respectively, with error handling for an empty queue.  
# - \*\*size\*\*: Returns the number of elements currently in the queue.