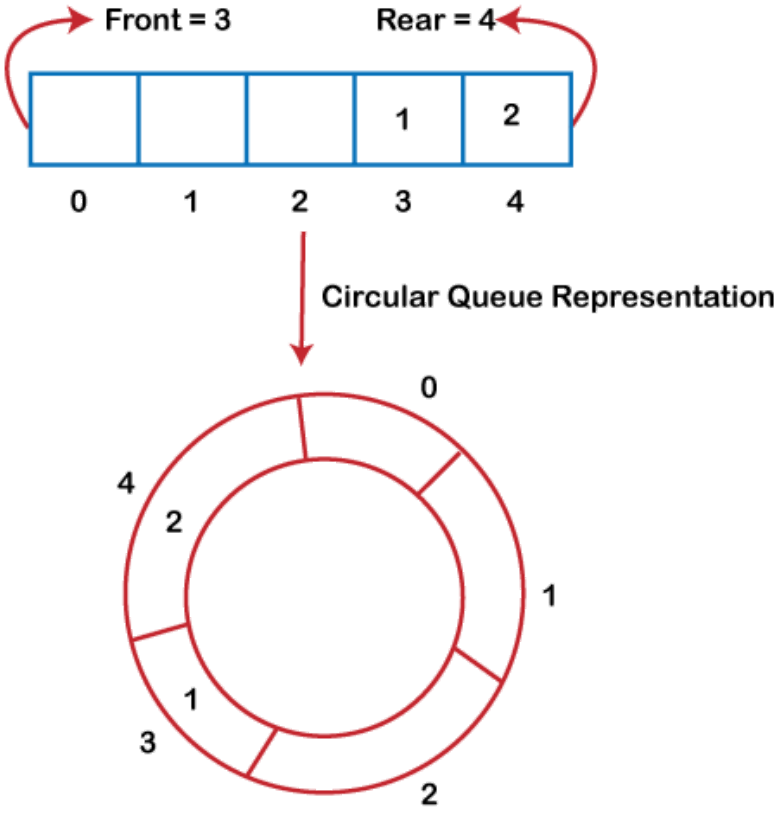
CS201 HW8

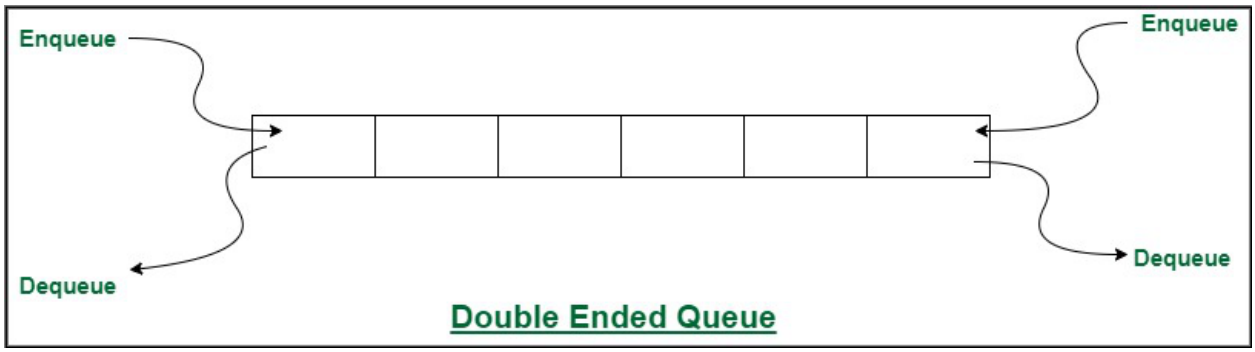
1. Describe different categories of queue with diagrams and possible operations.

A queue is a linear structure that follows a particular order in which the operations are performed. The order is First In First Out (FIFO). The three variations of a queue which add functionality to a regular queue that we will be discussing are the circular queue, the double ended queue and the priority queue. Input and output restricted queues are another two variations of the queue but we won't discuss them in as much detail.

- **Circular queue:** this data structure is first in first out (FIFO) like a regular queue, but the tail element of the structure points back to the head, creating a circle. In a normal queue the pointer of the tail element is set to null to signify the end of the structure, but a circular queue signifies the final element of the queue by setting its pointer to the head element. This type of data structure is advantageous for processes which require iterating over the elements in the structure repeatedly, since the circular structure makes this possible without having to reset the process as you would with a normal queue. Circular queues provide superior memory management compared to a normal queue since the unused memory locations in the case of ordinary queues can be utilized in circular queues.



- **Double Ended Queue (Dequeue):** This data structure functions as a versatile queue variation where insertion and deletion operations occur at both the front and rear ends. This implies the ability to add and remove elements from both the front and rear positions, meaning the head of the dequeue is much easier to access or pop compared to a normal queue. This functionality also allows a deque to serve as both a stack and queue. Given its support for both stack and queue operations, a deque serves a dual purpose. Notably, it enables clockwise and anticlockwise rotations in constant time (O(1)), which proves beneficial in specific applications. A benefit of the deque is that problems which require efficient removal and/or addition of elements at both ends of the data structure find effective solutions through the utilization of a deque.



- **Priority Queue:** this data structure is another variation of a typical queue in which each individual element is associated with a ‘priority’, and is served within the queue in relation to its priority. Elements can be inserted in any order and will be reformatted in order of priority. This grants the priority queue a time complexity of O(log n). Priority queues can be split into two subcategories based on whether the highest or lowest priority elements are placed at the tail, and thus removed first.
    - **Ascending priority queue:** highest priority elements start at the head, meaning elements will be removed from the structure in order from lowest priority to highest
    - **Descending priority queue:** lowest priority elements start at the head, meaning elements will be removed from the structure in order from highest priority to lowest

Descending Order Priority Queue

45	35	20	12	8	4

✓ In the above table, **4** has the **lowest priority**, and **45** has the **highest priority**.

Ascending Order Priority Queue

Example

4	8	12	20	35	45

✓ In the above table, **4** has the **highest priority**, and **45** has the **lowest priority**.

2. Write python programs to implement the three queue variants listed above with operations. (programs for each present in circular-queue.py, deque.py, and priority-queue.py) [output for each attached below]

```
nayatrov@ariahas-MBP py-hw % python3 circular-queue.py
Enqueuing elements into the circular queue:
Enqueued: 1
Enqueued: 2
Enqueued: 3
Enqueued: 4
Enqueued: 5
Is the circular queue empty? False
Circular Queue size: 5
Front element: 1
Dequeuing elements:
1
2
3
4
5
Circular Queue size: 0
```

```
nayatrov@ariahas-MBP py-hw % python3 deque.py
Enqueuing elements into the deque:
Is the deque empty? False
Deque size: 4
Front element: 2
Rear element: 4
Dequeuing elements from the front:
Dequeuing front: 2
Dequeuing front: 1
Dequeuing front: 3
Dequeuing front: 4
Deque size: 0
Enqueuing more elements into the deque:
Front element: 5
Rear element: 6
Dequeuing elements from the rear:
Dequeuing rear: 6
Dequeuing rear: 5
Deque size: 0
```

```
nayatrov@ariahas-MBP py-hw % python3 priority-queue.py
Enqueuing elements into the priority queue:
Task 1: priority 3
Task 2: priority 1
Task 3: priority 2
Is the priority queue empty? False
Priority Queue size: 3
Front element: Task 2
Dequeuing elements:
Task 2
Task 3
Task 1
Priority Queue size: 0
```

3. Describe the limitations of a queue data structure.

A normal queue data structure has the limitation of a fixed size and a linear structure for both enqueue and dequeue operations, which makes the variants talked about in this assignment more efficient for various reasons. Here are some limitations of a normal

queue compared to the three variants implemented and described above (Circular Queue, Priority Queue, and Deque):

- **Lack of ordering / prioritization:** a normal queue orders elements only by the order they are added to the queue. This makes a priority queue superior for any tasks which benefit from ordering.
- **Limited access to elements:** a normal queue can only remove elements in first in first out (FIFO) order, meaning if we want to access the head of the queue, we must first remove all other elements in the structure. This makes a deque superior for any tasks which benefit from being able to remove elements from both the head and tail of the queue.
- **Fixed size:** a normal queue has a fixed size, and can have stack overflow errors if elements are attempted to be added after the queue is full. This makes a circular queue superior for any tasks in which a fixed size data structure is not optimal.

While a normal queue is suitable for many scenarios, these three variants (Circular Queue, Priority Queue, and Deque) address specific limitations and offer enhanced functionality to better suit different use cases. The variants are not always superior to a regular queue, and can in fact be worse than a normal queue in certain conditions. Thus the choice of the data structure depends on the requirements and characteristics of the problem being solved.