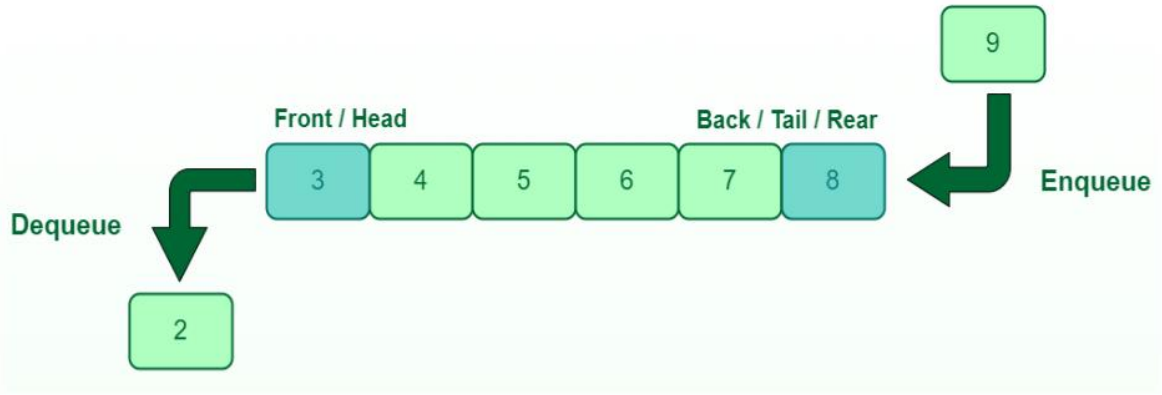
What is Queue Data Structure?

A **Queue** is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.



Queue Data Structure

- **Queue:** the name of the array storing queue elements.
- **Front**: the index where the first element is stored in the array representing the queue.
- **Rear:** the index where the last element is stored in an array representing the queue.

Queue Implementation Using Array:

- **Class Definition:**

  The code defines a C++ class named Queue, representing a queue data structure.

- **Class Members:**

  int front and int rear: These variables represent the indices of the front and rear elements of the queue, respectively. When both are -1, it indicates that the queue is empty.

  int size: Stores the maximum size of the queue.

  int *arr: Dynamically allocated integer array used to store the elements of the queue.

- **Inside the constructor:**

  front and rear are initialized to -1 to indicate an empty queue.

  The size is set to the specified maximum size.

Memory for the queue is dynamically allocated using the new keyword.

- **isEmpty() Method:**

  bool isEmpty(): Checks if the queue is empty.

  Returns true if both front and rear are -1, indicating an empty queue; otherwise, it returns false.

- **isFull() Method:**

  bool isFull(): Checks if the queue is full.

  Returns true if the next index after rear (with wrapping) equals front, indicating a full queue; otherwise, it returns false.

- **enqueue(int val) Method:**

  Adds an element with the value val to the rear of the queue.

  Checks if the queue is full using the isFull() method:

  If the queue is full, it prints "Queue is full" and returns without enqueueing.

  If the queue is empty initially, it sets both front and rear to 0.

  Otherwise, it increments rear while taking into account circular wrapping.

  The val is added to the arr at the rear index, and an "Enqueued: <val>" message is printed.

- **dequeue() Method:**

  Removes an element from the front of the queue.

  Checks if the queue is empty using the isEmpty() method:

  If the queue is empty, it prints "Queue is empty" and returns without dequeueing.

  If there's only one element in the queue, it resets both front and rear to -1. Representing that the only element is removed and now the front and rear has reset again.

  Otherwise, it increments front while considering circular wrapping.

  The dequeued value is stored in val, and a "Dequeued: <val>" message is printed.

- **getFront() Method:**

  Returns the element at the front of the queue without removing it. It simply retrieves the value from the arr at the front index.

- **display() Method:**

Displays all elements in the queue, starting from the front and wrapping around if necessary.

It iterates through the elements in the circular queue and prints them one by one.

```
#include<iostream>
using namespace std;
class Queue {
		int front;
		int rear;
		int size;
		int *arr;
	public:
		Queue(int size) {
			front=-1;
			rear=-1;
			this->size=size;
			arr=new int[size];
		}
		bool isEmpty() {
			if(front==-1 && rear==-1) {
				return true;
			} else {
				return false;
			}
		}
		bool isFull() {
			if(((rear+1)%size) == front) {
				return true;
			} else {
				return false;
			}
		}
```

```
        void enqueue(int val) {
            if(isFull()) {
                cout<<"Queue is full"<<endl;
                return;
            } else if(isEmpty()) {
                front=0;
                rear=0;
            } else {
                rear=(rear+1)%size;
            }
            arr[rear]=val;
            cout<<"Enqueued: "<<val<<endl;
        }
        void dequeue() {
            if(!isEmpty()) {
                int val=arr[front];
                if(front==rear) {
                    front=-1;
                    rear=-1;
                } else {
                    front=(front+1)%size;
                }
                cout<<"Dequeued: "<<val<<endl;
            } else {
                cout<<"Queue is empty"<<endl;
            }

        }

        int getFront() {
            int f = arr[front];
            cout<<"front: "<<f<<endl;
            return f;
        }
        int getRear() {
            int r = arr[rear];
            cout<<"rear: "<<r<<endl;
            return r;
        }

};
```

```
int main() {
        Queue q(5);
        q.enqueue(1);
        q.getFront();
        q.getRear();
        q.enqueue(2);
        q.getFront();
        q.getRear();
        q.enqueue(3);
        q.getFront();
        q.getRear();
        q.dequeue();
        q.getFront();
        q.getRear();
        q.dequeue();
        q.getFront();
        q.getRear();
        q.enqueue(6);
        q.getFront();
        q.getRear();
}
```

Output:

Enqueued: 1

front: 1

rear: 1

Enqueued: 2

front: 1

rear: 2

Enqueued: 3

front: 1

rear: 3

Dequeued: 1

front: 2

rear: 3

Dequeued: 2

front: 3

rear: 3

Enqueued: 6

front: 3

rear: 6

Implement the queue using linked list your self.