PW SKILLS

Lesson Plan

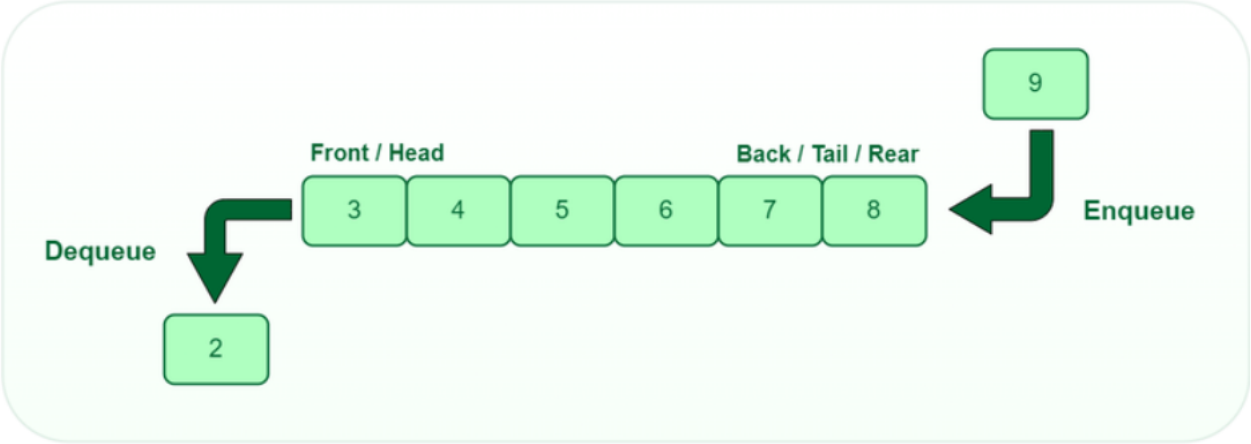
Queues 1

PW SKILLS

Today's checklist:

- Queue data structure
- Queue Vs Stack
- STL operations on queue
- Overflow and underflow
- Array implementation of queue
- Linkedlist implementation of queue
- Advantages of Linked list implementation of the queue over array implementation:
- Disadvantages of Linked list implementation of the queue over array implementation:
- Practice questions on Queue
- Dequeue
- Circular Queue
- Design Circular Queue [Leetcode - 622]

Queues are linear data structures that follow the First In, First Out (FIFO) principle, where the first element added to the queue is the first one to be removed. Think of it like a line at a ticket counter or a queue of cars waiting at a toll booth.



Queue vs Stack:

Queue: Elements are inserted at the rear (enqueue) and removed from the front (dequeue).
Stack: Elements are inserted (pushed) and removed (popped) from the same end, called the top.

STL Operations performed on queue:

In C++, the Standard Template Library (STL) provides a queue container that supports various operations:

push(): Adds an element to the back of the queue.
pop(): Removes the front element from the queue.
front(): Accesses the front element without removing it.
empty(): Checks if the queue is empty.
size(): Returns the number of elements in the queue.

Example:

```
#include <iostream>
#include <queue>

using namespace std;

int main() {
    queue<int> myQueue;

    myQueue.push(10);
    myQueue.push(20);
    myQueue.push(30);

    cout << "Front of the queue: " << myQueue.front() << endl;
    myQueue.pop();
    cout << "Front after pop: " << myQueue.front() << endl;

    cout << "Queue size: " << myQueue.size() << endl;

   if(myQueue.empty()) cout<<”empty queue”<<endl;
   else cout<<”non-empty queue”<<endl;
   return 0;
}
```

Overflow vs Underflow:

Overflow: This occurs when trying to add an element to a full data structure, like a queue or a stack.

Underflow: This occurs when trying to remove an element from an empty data structure, like a queue or a stack.

Q. Reverse the queue using a stack.

Input: 1,2,3,4,5
Output 5,4,3,2,1

Code:

```
void reverseQueue(queue<int> &q) {
    stack<int> s;

    while (!q.empty()) {
        s.push(q.front());
        q.pop();
    }

    while (!s.empty()) {
        q.push(s.top());
        s.pop();
    }
}
```

Time: O(n), q.pop() is O(1) we do it n times for n elements
Space: O(n), for extra stack used

Explanation: We can keep popping the first element from queue and pushing it in the stack now stack will have 5 on top than 4 so on at bottom 1 now keeping pushing the top of stack to the queue , thus our queue is reversed

Q. Remove all the elements present at even positions in queue. Consider 0-based indexing.

Input Queue: [10, 20, 30, 40, 50, 60, 70]
Output: Queue after removing elements at even positions: [10, 30, 50, 70]

Code:

```
void removeEvenPositions(queue<int> &q) {
    queue<int> tempQueue;

    int index = 0;
    while (!q.empty()) {
        if (index % 2 =≠ 0) {
            tempQueue.push(q.front());
        }
        q.pop();
        index++;
    }

    q = tempQueue; // Copy back the modified elements to the
original queue
}
```

Time: O(n), to traverse the queue
Space: O(n), for the temp queue

Explanation: Create an index variable and temp queue, traverse the queue, and keep increasing the index, when at an odd index push the element in the temp queue otherwise just pop
At last set, the queue equals to temp queue which only has elements at odd indexes

Array implementation Of Queue:

For implementing a queue, we need to keep track of two indices, front and rear. We enqueue an item at the rear and dequeue an item from the front. If we simply increment front and rear indices, then there may be problems, the front may reach the end of the array. The solution to this problem is to increase the front and rear in a circular manner.

Steps for enqueue:

Check the queue is full or not
If full, print overflow and exit
If queue is not full, increment tail and add the element

Steps for dequeue:

Check queue is empty or not
if empty, print underflow and exit
if not empty, print element at the head and increment head

Below is a program to implement above operation on queue

Code:

```
#include <iostream>
using namespace std;

class Queue {
public:
    int front, rear, size;
    unsigned capacity;
    int* array;
};

Queue* createQueue(unsigned capacity) {
    Queue* queue = new Queue();
    queue->capacity = capacity;
    queue->front = queue->size = 0;
    queue->rear = capacity - 1;
    queue->array = new int[queue->capacity];
    return queue;
}

int isFull(Queue* queue) {
    return (queue->size == queue->capacity);
}

int empty(Queue* queue) {
    return (queue->size == 0);
}

void push(Queue* queue, int item) {
    if (isFull(queue))
        return;
    queue->rear = (queue->rear + 1) % queue->capacity;
    queue->array[queue->rear] = item;
    queue->size = queue->size + 1;
    cout << item << " pushed to queue\n";
}

int pop(Queue* queue) {
    if (empty(queue))
        return INT_MIN;
    int item = queue->array[queue->front];
    queue->front = (queue->front + 1) % queue->capacity;
    queue->size = queue->size - 1;
    return item;
}
```

```
int front(Queue* queue) {
    if (empty(queue))
        return INT_MIN;
    return queue->array[queue->front];
}

int rear(Queue* queue) {
    if (empty(queue))
        return INT_MIN;
    return queue->array[queue->rear];
}

int size(Queue* queue) {
    return queue->size;
}


void display(Queue* queue) {
    int i = queue->front;
    while (i != queue->rear) {
        cout << queue->array[i] << " ";
        i = (i + 1) % queue->capacity;
    }
    cout << queue->array[i] << " ";
}

int main() {
    Queue* queue = createQueue(1000);

    push(queue, 10);
    push(queue, 20);
    push(queue, 30);
    push(queue, 40);

    cout << pop(queue) << " popped from queue\n";

    cout << "Front item is " << front(queue) << endl;
    cout << "Rear item is " << rear(queue) << endl;

   cout << "Queue size is " << size(queue) << endl;


    cout << "Queue elements are: ";
    display(queue);

    return 0;
}
```

Explanation: createQueue(unsigned capacity):

Initializes a queue structure by allocating memory for an array to hold elements, setting the front, rear, and size pointers to manage elements, based on the specified capacity.
isFull(Queue* queue):

Check if the current size of the queue has reached its maximum capacity, ensuring that no more elements can be added.
empty(Queue* queue):

Verifies if the queue size is zero, indicating there are no elements present in the queue.
push(Queue* queue, int item):

Adds an element to the queue by incrementing the rear index (circularly, if needed) and storing the item in the array at that index.
pop(Queue* queue):

Removes an element from the queue by incrementing the front index (circularly, if needed) and reducing the queue size.
front(Queue* queue):

Retrieves the element at the front of the queue by accessing the array element pointed to by the front index.
rear(Queue* queue):

Retrieves the element at the rear of the queue by accessing the array element pointed to by the rear index.
display(Queue* queue):

Iterates through the queue array, starting from the front index, and prints each element until reaching the rear index, displaying all elements of the queue.

Complexity Analysis:
Time Complexity

Operations	Complexity
• push(insertion)	O(1)
• pop(deletion)	O(1)
• Front(Get front)	O(1)
• Rear(Get Rear)	O(1)
• size(to get q's size)	O(1)
• IsFull(Check queue is full or not)	O(1)
• IsEmpty(Check queue is empty or not)	O(1)
• Display:	O(n), since we traverse the whole queue

Auxiliary Space:
O(N) where N is the size of the array for storing elements.

LinkedList implementation of Queue

Explanation:
we maintain two pointers, front, and rear. The front points to the first item of the queue and rear points to the last item.

enQueue(): This operation adds a new node after the rear and moves the rear to the next node.

deQueue(): This operation removes the front node and moves the front to the next node.

Follow the below steps to implement a queue using LinkedList:

Create a class QNode with data members integer data and QNode* next

- A parameterized constructor that takes an integer x value as a parameter and sets data equal to x and next as NULL

Create a class Queue with data members QNode front and rear
Enqueue Operation with parameter x:

- Initialize QNode* temp with data = x
- If the rear is set to NULL then set the front and rear to temp and return(Base Case)
- Else set rear next to temp and then move rear to temp

Dequeue Operation:

- If the front is set to NULL return(Base Case)
- Initialize QNode temp with front and set front to its next
- If the front is equal to NULL then set the rear to NULL
- Delete temp from the memory

Code:

```
#include <bits/stdc++.h>
using namespace std;

struct QNode {
    int data;
    QNode* next;
    QNode(int d) {
        data = d;
        next = NULL;
    }
};

struct Queue {
    QNode *front, *rear;
    Queue() { front = rear = NULL; }

    void enQueue(int x) {
        QNode* temp = new QNode(x);
        if (rear == NULL) {
            front = rear = temp;
            return;
        }
```

```
        rear->next = temp;
        rear = temp;
    }

    void deQueue() {
        if (front == NULL)
            return;
        QNode* temp = front;
        front = front->next;
        if (front == NULL)
            rear = NULL;
        delete (temp);
    }

    int getSize() {
        int count = 0;
        QNode* current = front;
        while (current != NULL) {
            count++;
            current = current->next;
        }
        return count;
    }

    void display() {
        if (front == NULL) {
            cout << "Queue is empty" << endl;
            return;
        }
        QNode* current = front;
        while (current != NULL) {
            cout << current->data << " ";
            current = current->next;
        }
        cout << endl;
    }
};

int main() {
    Queue q;
    q.enQueue(10);
    q.enQueue(20);
    q.deQueue();
    q.deQueue();
    q.enQueue(30);
    q.enQueue(40);
    q.enQueue(50);

    cout << "Queue Front : " << ((q.front != NULL) ? (q.front)
->data : -1) << endl;
```

```
    cout << "Queue Rear : " << ((q.rear != NULL) ? (q.rear)->data
: -1) << endl;
    cout << "Queue Size : " << q.getSize() << endl;
    cout << "Queue Elements: ";
    q.display();

    return 0;
}
```

Operations on Linked List-based Queue:

enQueue(int x):

Time Complexity: O(1)

Explanation: Inserts an element at the rear of the queue in constant time by updating the rear pointer.

deQueue():

Time Complexity: O(1)

Explanation: Removes an element from the front of the queue in constant time by updating the front pointer.

getSize():

Time Complexity: O(n)

Explanation: Iterates through the entire queue to count and return the number of elements, where 'n' is the number of elements in the queue.

display():

Time Complexity: O(n)

Explanation: Traverses through the entire queue to print all elements, where 'n' is the number of elements in the queue.

The enQueue and deQueue operations execute in constant time since they involve updating front and rear pointers. However, getSize and display iterate through the queue, leading to linear time complexity proportional to the number of elements in the queue.

Advantages of Linked list implementation of the queue over array implementation:

- **Dynamic Sizing:** Linked lists allow the queue to adjust its size dynamically, accommodating varying numbers of elements without requiring a predefined capacity.
- **Efficient Memory Utilization:** Memory is allocated dynamically as elements are added, utilizing only the necessary space, preventing wasted memory compared to fixed-size arrays.

- **Efficient Insertion and Deletion:** Adding or removing elements at the front or rear of the queue is efficient in linked lists, avoiding the need to shift elements as in array implementations.
- **Avoidance of Overflow:** Linked lists prevent overflow issues that can occur in arrays by dynamically allocating memory.
- **No Reallocation Overhead:** They eliminate the need for reallocation or resizing when the queue size changes, unlike arrays, which might require resizing operations.
- **Flexible Implementation:** Linked lists offer flexibility, enabling easy modifications like inserting or removing nodes without dealing with array reallocation complexities. This flexibility suits scenarios with varying or unpredictable queue sizes.

Disadvantages of Linked list implementation of the queue over array implementation:

- **Higher Memory Overhead:** Linked lists use additional memory per element to store pointers/references, leading to higher memory overhead compared to arrays, which only store the data elements themselves.
- **Less Cache-Friendly:** Linked lists don't offer contiguous memory allocation like arrays, which can result in less cache-friendly behavior, impacting performance due to frequent jumps in memory locations.
- **Slower Access Time:** Accessing elements in a linked list-based queue can be slower compared to arrays, especially for random access or traversal operations, due to pointer-based navigation.
- **More Complex Memory Management:** Managing memory in a linked list involves dynamic allocation and deallocation of memory for each node, potentially leading to more complex memory management compared to arrays.
- **Inefficient for Some Operations:** Certain operations, such as accessing elements at specific indices or range-based operations, are less efficient in linked lists compared to arrays due to the lack of direct access.
- **Less Predictable Performance:** Linked list performance can be less predictable in terms of memory access patterns and can suffer from cache misses, impacting performance in certain scenarios compared to arrays with better cache utilization.

In summary, while linked lists offer flexibility, dynamic sizing, and ease of insertion/deletion, they can suffer from higher memory overhead, slower access times, and less cache-friendly behavior compared to arrays, especially in scenarios where direct access to elements or memory efficiency is critical.

Dequeue:

Deque or Double Ended Queue is a generalized version of the Queue data structure that allows insert and delete at both ends

Implementation of Dequeue:

```
#include <iostream>
using namespace std;

struct Node {
```

```
    int data;
    Node* next;
    Node* prev;
    Node(int val) : data(val), next(nullptr), prev(nullptr) {}
};

class Deque {
private:
    Node* front;
    Node* rear;
    int count;

public:
    Deque() : front(nullptr), rear(nullptr), count(0) {}

    void addFront(int val) {
        Node* newNode = new Node(val);
        if (front == nullptr) {
            front = rear = newNode;
        } else {
            newNode->next = front;
            front->prev = newNode;
            front = newNode;
        }
        count++;
    }

    void addRear(int val) {
        Node* newNode = new Node(val);
        if (rear == nullptr) {
            front = rear = newNode;
        } else {
            rear->next = newNode;
            newNode->prev = rear;
            rear = newNode;
        }
        count++;
    }

    int getFront() {
        return (front != nullptr) ? front->data : -1; // Return
-1 for an empty deque
    }

    int getRear() {
        return (rear != nullptr) ? rear->data : -1; // Return -1
for an empty deque
    }

    void deleteFront() {
        if (front != nullptr) {
```

```
            Node* temp = front;
            front = front->next;
            if (front != nullptr) {
                front->prev = nullptr;
            } else {
                rear = nullptr;
            }
            delete temp;
            count--;
        }
    }

    void deleteRear() {
        if (rear != nullptr) {
            Node* temp = rear;
            rear = rear->prev;
            if (rear != nullptr) {
                rear->next = nullptr;
            } else {
                front = nullptr;
            }
            delete temp;
            count--;
        }
    }

    int size() {
        return count;
    }
};

int main() {
    Deque dq;

    dq.addRear(10);
    dq.addRear(20);
    dq.addFront(5);

    cout << "Front element: " << dq.getFront() << endl; //
Output: 5
    cout << "Rear element: " << dq.getRear() << endl; // Output:
20

    dq.deleteFront();
    dq.deleteRear();

    cout << "Size: " << dq.size() << endl; // Output: 1

    return 0;
}
```

Explanation:

- addFront: Inserts an element at the front of the deque.
- addRear: Inserts an element at the rear of the deque.
- getFront: Retrieves the front element of the deque.
- getRear: Retrieves the rear element of the deque.
- deleteFront: Removes the front element of the deque.
- deleteRear: Removes the rear element of the deque.
- size: Returns the current size of the deque.

This implementation uses a doubly linked list to support insertion and deletion at both ends of the deque, providing functionalities to interact with the front, rear elements, and the size of the deque.

Time Complexity:

- **addFront(int val):**
- **Time Complexity: O(1)**
- **Explanation:** Adding an element at the front involves creating a new node and adjusting pointers, all of which are constant-time operations.
- **addRear(int val):**
- **Time Complexity: O(1)**
- **Explanation:** Adding an element at the rear also involves creating a new node and adjusting pointers in constant time.
- **getFront():**
- **Time Complexity: O(1)**
- **Explanation:** Retrieving the front element directly involves accessing the front node's data, which is a constant-time operation.
- **getRear():**
- **Time Complexity: O(1)**
- **Explanation:** Retrieving the rear element similarly involves accessing the rear node's data, also a constant-time operation.
- **deleteFront():**
- **Time Complexity: O(1)**
- **Explanation:** Deleting the front element involves adjusting pointers to the next node, which is a constant-time operation.
- **deleteRear():**
- **Time Complexity: O(1)**
- **Explanation:** Deleting the rear element involves adjusting pointers to the previous node, also a constant-time operation.
- **size():**
- **Time Complexity: O(1)**
- **Explanation:** Maintaining the count of elements allows direct retrieval of the size in constant time without iteration through the deque.

Space Complexity:

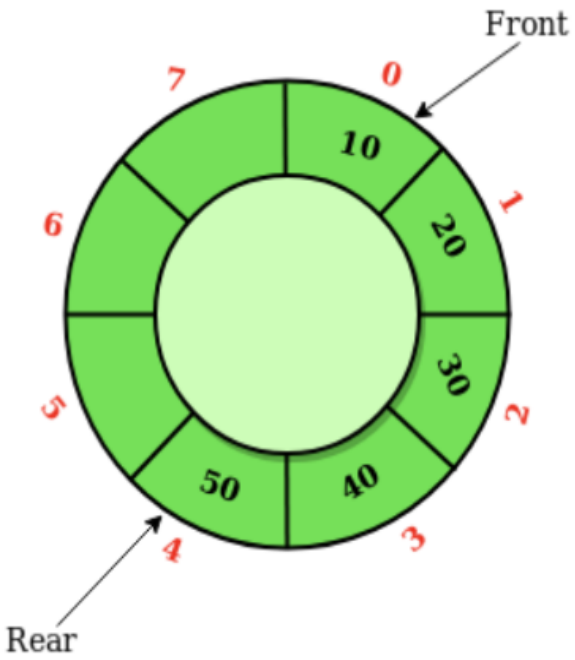
- **Nodes:**
- **Space Complexity: O(n)**
- **Explanation:** Each element in the deque is stored in a node that contains the data and two pointers (prev and next). For 'n' elements, 'n' nodes are created, resulting in space proportional to the number of elements.

- **Other Variables:**
- **Space Complexity: O(1)**
- **Explanation:** Additional pointers (front, rear) and a counter (count) used to maintain the structure take constant space regardless of the number of elements.

Circular Queue:

A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

The operations are performed based on FIFO (First In First Out) principle. It is also called 'Ring Buffer'.
In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.



Operations on Circular Queue:

Front: Get the front item from the queue.
Rear: Get the last item from the queue.
enQueue(value) This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.
Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
If it is full then display Queue is full.
If the queue is not full then, insert an element at the end of the queue.
deQueue() This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.

Check whether the queue is Empty.
If it is empty then display Queue is empty.
If the queue is not empty, then get the last element and remove it from the queue.

Q. Design Circular Queue [Leetcode - 622]

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implement the MyCircularQueue class:

MyCircularQueue(k) Initializes the object with the size of the queue to be k.
int Front() Gets the front item from the queue. If the queue is empty, return -1.
int Rear() Gets the last item from the queue. If the queue is empty, return -1.
boolean enQueue(int value) Inserts an element into the circular queue. Return true if the operation is successful.
boolean deQueue() Deletes an element from the circular queue. Return true if the operation is successful.
boolean isEmpty() Checks whether the circular queue is empty or not.
boolean isFull() Checks whether the circular queue is full or not.
You must solve the problem without using the built-in queue data structure in your programming language.

Example 1:

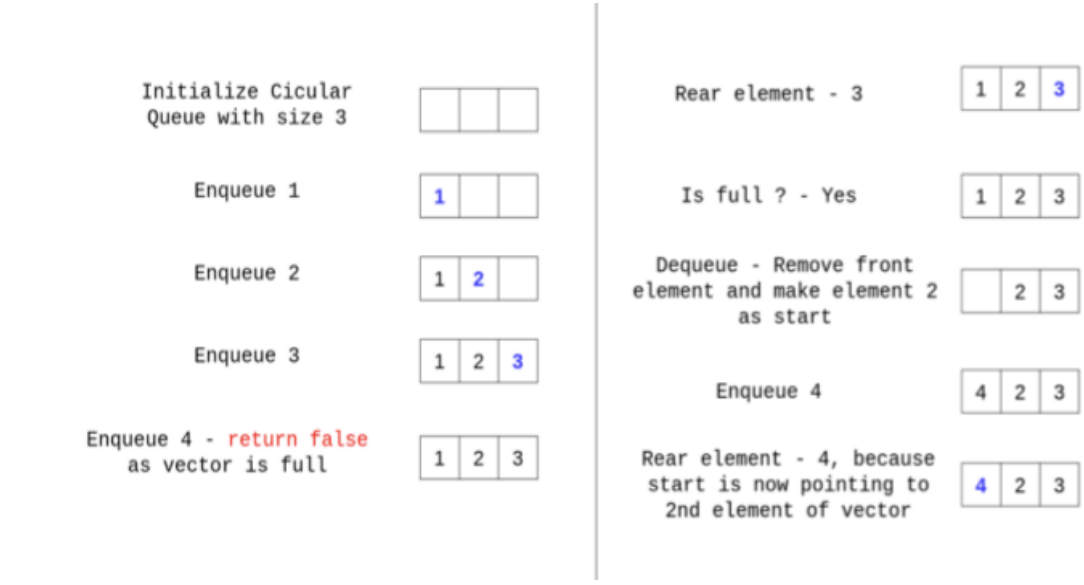
Input:
["MyCircularQueue", "enQueue", "enQueue", "enQueue", "enQueue", "Rear", "isFull", "deQueue", "enQueue", "Rear"]
[[3], [1], [2], [3], [4], [], [], [], [4], []]
Output:
[null, true, true, true, false, 3, true, true, true, 4]

Explanation:

MyCircularQueue myCircularQueue = new MyCircularQueue(3);
myCircularQueue.enQueue(1); // return True
myCircularQueue.enQueue(2); // return True
myCircularQueue.enQueue(3); // return True
myCircularQueue.enQueue(4); // return False
myCircularQueue.Rear(); // return 3
myCircularQueue.isFull(); // return True
myCircularQueue.deQueue(); // return True
myCircularQueue.enQueue(4); // return True
myCircularQueue.Rear(); // return 4

Code:

```
class MyCircularQueue {
private:
    vector<int> v;
    int start = 0, len = 0;
public:
    MyCircularQueue(int k): v(k) {}
    bool enQueue(int value) {
        if (isFull()) return false;
        v[(start + len++) % v.size()] = value;
        return true;
    }
    bool deQueue() {
        if (isEmpty()) return false;
        start = (start + 1) % v.size();
        --len;
        return true;
    }
    int Front() {
        if (isEmpty()) return -1;
        return v[start];
    }
    int Rear() {
        if (isEmpty()) return -1;
        return v[(start + len - 1) % v.size()];
    }
    bool isEmpty() {
        return !len;
    }
    bool isFull() {
        return len == v.size();
    }
};
```



Explanation:

Create a vector and two variable start and len.
When we insert something in queue, increase the len.
When we remove something from queue, decrease the len.

Time Complexity:

Enqueue: O(1) because no loop is involved for a single enqueue.
Dequeue: O(1) because no loop is involved for one dequeue operation.
Similarly front, rear, isEmpty, isFull all are O(1)

Auxiliary Space: O(N) as the queue is of size N.

PW SKILLS

THANK YOU !

PW SKILLS