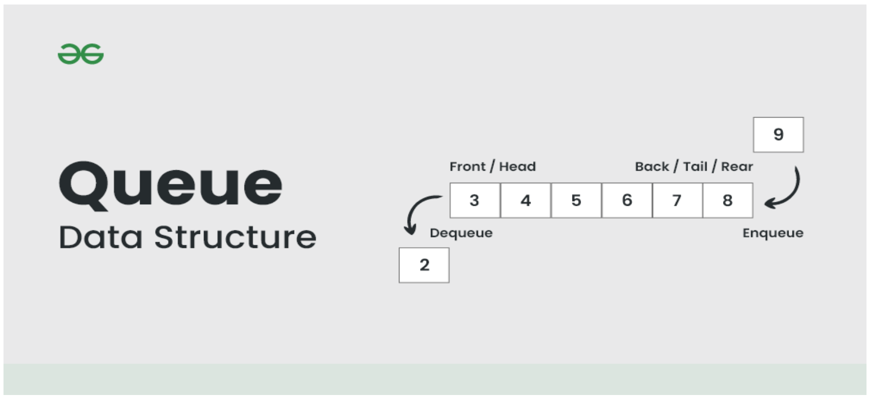
Lab 4

Implementation of Queue

A Queue Data Structure is a fundamental concept in computer science used for storing and managing data in a specific order. It follows the principle of "First in, First out" (FIFO), where the first element added to the queue is the first one to be removed. Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow



Queue ADT Operations

- Initialize -- Sets queue to an empty state.
- IsEmpty -- Determines whether the queue is currently empty.
- IsFull -- Determines whether the queue is currently full.
- Insert (ItemType newItem) -- Adds newItem to the rear of the queue.
- Remove (ItemType& item) -- Removes the item at the front of the queue and returns it in item.

Implementation of Queue Using Circular Arrays

Given

- an array **Items**[0:N-1] consisting of N items
- two indices **Front** and **Rear**, that designate positions in the Items array

We can use the following assignments to increment the indices so that they always wrap around after falling off the high end of the array.

front = (front + 1) % N
rear = (rear + 1) % N

```
//_______________________________________
// CLASS DEFINITION FOR QUEUE
//_______________________________________
#include <iostream>
#define maxQue 100
typedef int ItemType;

class Queue {
private:
ItemType items[maxQue];  // Array to store queue
items
     int front, rear, count;  // Front and rear pointers, and count of items

public:
  Queue();             // Constructor
     int IsEmpty();         // Check if the queue is empty
     int IsFull();          // Check if the queue is full
    void Insert(ItemType newItem);  // Insert a new item into the queue
     void Remove(ItemType &item);   // Remove an item from the queue
};

// Constructor: Initializes an empty queue
Queue::Queue() {
  count = 0;
  front = 0;
  rear = 0;
}

// Check if the queue is empty
int Queue::IsEmpty() {
  return (count == 0);
}

// Check if the queue is full
int Queue::IsFull() {
  return (count == maxQue);
}
```

```
// Insert a new item into the queue
void Queue::Insert(ItemType newItem) {
    if (IsFull()) {
        std::cout << "Overflow: Cannot insert,
queue is full.\n";
    }

 else {
        items[rear] = newItem;
        rear = (rear + 1) % maxQue;  // Circular
queue logic
        ++count;
    }
}

// Remove an item from the queue
void Queue::Remove(ItemType &item) {
    if (IsEmpty()) {
        std::cout << "Underflow: Cannot remove,
queue is empty.\n";
    } else {
        item = items[front];
        front = (front + 1) % maxQue;  // Circular
queue logic
        --count;
    }
}

int main() {
    Queue q;
    ItemType item;

    // Test inserting items into the queue
    for (int i = 1; i <= 5; ++i) {
        q.Insert(i);
        std::cout << "Inserted: " << i << std::endl;
    }

    // Test removing items from the queue
    while (!q.IsEmpty()) {
        q.Remove(item);
        std::cout << "Removed: " << item <<
std::endl;
```

```
    }

    return 0;
}
```

Output:

```
Inserted: 1
Inserted: 2
Inserted: 3
Inserted: 4
Inserted: 5
Removed: 1
Removed: 2
Removed: 3
Removed: 4
Removed: 5
```

Dynamic Implementation of Queue

Queue Using Template and Dynamic Array

```
//______________________________________________
// CLASS TEMPLATE DEFINITION FOR QUEUE
//______________________________________--------------

#include <iostream>
using namespace std;

template<class ItemType>
class Que {
public:
    Que();                    // Default constructor
    Que(int max);               // Parameterized constructor
    ~Que();                   // Destructor

    int IsFull() const;        // Check if the queue is full
    int IsEmpty() const;       // Check if the queue is empty

    void Insert(ItemType newItem);  // Insert a new item into the queue
    void Remove(ItemType& item);    // Remove an item from the queue

private:
    int front;               // Front index
    int rear;                // Rear index
    int maxQue;                // Maximum size of the queue (array size)
    int count;               // Current number of elements in the queue
    ItemType* items;           // Dynamic array for storing items
};

// Default constructor
template<class ItemType>
```

```
Que<ItemType>::Que() {
    maxQue = 501;              // Default size of the queue
    front = 0;
    rear = 0;
    count = 0;
    items = new ItemType[maxQue];  // Dynamically allocate array
}

// Parameterized constructor
template<class ItemType>
Que<ItemType>::Que(int max) {
    maxQue = max + 1;          // Set maxQue to max + 1 for circular logic
    front = 0;
    rear = 0;
    count = 0;
    items = new ItemType[maxQue];  // Dynamically allocate array
}

// Destructor
template<class ItemType>
Que<ItemType>::~Que() {
    delete[] items;            // Deallocate the dynamic array
}

// Check if the queue is empty
template<class ItemType>
int Que<ItemType>::IsEmpty() const {
    return (count == 0);
}

// Check if the queue is full
template<class ItemType>
int Que<ItemType>::IsFull() const {
    return (count == maxQue - 1);  // Queue is full if count == maxQue - 1
}

// Insert a new item into the queue
template<class ItemType>
void Que<ItemType>::Insert(ItemType newItem) {
    if (IsFull()) {
        cout << "Overflow: Cannot insert, queue is full.\n";
    } else {
        items[rear] = newItem;              // Add item to the rear
        rear = (rear + 1) % maxQue;         // Circular increment
        ++count;
    }
}

// Remove an item from the queue
template<class ItemType>
void Que<ItemType>::Remove(ItemType& item) {
    if (IsEmpty()) {
        cout << "Underflow: Cannot remove, queue is empty.\n";
    } else {
        item = items[front];                // Remove item from the front
        front = (front + 1) % maxQue;       // Circular increment
        --count;
    }
}
```

```
// Example driver program
int main() {
    Que<int> q(5);  // Create a queue with a capacity of 5 elements
    int item;

    // Insert elements into the queue
    q.Insert(10);
    q.Insert(20);
    q.Insert(30);
    q.Insert(40);
    q.Insert(50);

    // Try inserting when the queue is full
    q.Insert(60);

    // Remove elements from the queue
    while (!q.IsEmpty()) {
        q.Remove(item);
        cout << "Removed: " << item << endl;
    }

    // Try removing when the queue is empty
    q.Remove(item);

    return 0;
}
```

Output:

```
Overflow: Cannot insert, queue is full.
Removed: 10
Removed: 20
Removed: 30
Removed: 40
Removed: 50
Underflow: Cannot remove, queue is empty.
```

Task 1:

Write a driver program to insert 10 numbers in a queue and then remove and print the numbers

Hint:

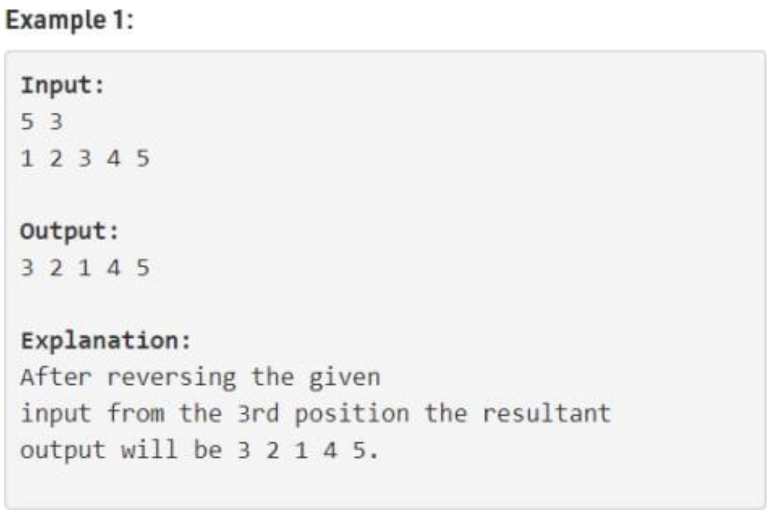
• **Inserting numbers:** Numbers from 1 to 10 are inserted into the queue using `push()`. During each insertion, both `front()` and `back()` operations are used to display the current front and back elements of the queue.

• **Removing and printing:** The program removes each number using `pop()`. Before each removal, the `front ()` and `back()` operations are used to display the first and last elements in the queue at that moment.

Task 2:

Reverse First K elements of Queue Given an integer K and a queue of integers, we need to reverse the order of the first K elements of the queue, leaving the other elements in the same relative order. Only following standard operations are allowed on queue.

- **enqueue(x)** : Add an item x to rear of queue
- **dequeue()** : Remove an item from front of queue
- **size()** : Returns number of elements in queue.
- **front()** : Finds front item.



Example 1:

```
Input:
5 3
1 2 3 4 5

Output:
3 2 1 4 5

Explanation:
After reversing the given
input from the 3rd position the resultant
output will be 3 2 1 4 5.
```

Hint:

1. **Initialize the queue** and insert 8 numbers into it.
2. **Check if `K` is valid** before proceeding with the reversal.
3. **Remove the first `K` elements** and store them in an array.
4. **Insert the elements in reverse order** back into the queue.
5. **Rearrange the remaining elements** by dequeuing and re-enqueuing them to maintain their order.
6. **Print the final queue** after the reversal