AIM-

To study the concept of simple queue.

Theory of Simple Queue

A **queue** is a linear data structure that follows the First In, First Out (FIFO) principle. This means that the first element added to the queue will be the first one to be removed. Queues are commonly used in scenarios such as scheduling processes in operating systems, handling requests in web servers, and implementing breadth-first search algorithms.

Key Operations of a Queue

1. **Enqueue**: Add an element to the back of the queue.
2. **Dequeue**: Remove and return the front element of the queue.
3. **Front/Peek**: Get the front element without removing it from the queue.
4. **isEmpty**: Check if the queue is empty.
5. **isFull**: Check if the queue is full (if implemented with a fixed size).

Queue Implementation

Queues can be implemented using arrays or linked lists. Here, we'll provide an implementation using both methods in C.

Implementation Using an Array

```
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

#define MAX 100 // Maximum size of the queue

typedef struct Queue {
    int front, rear, size;
    int arr[MAX];
} Queue;

// Function to create and initialize the queue
Queue* createQueue() {
    Queue* queue = (Queue*)malloc(sizeof(Queue));
    queue->front = 0;
    queue->rear = -1;
    queue->size = 0;
    return queue;
```

```
}

// Check if the queue is empty
int isEmpty(Queue* queue) {
    return queue->size == 0;
}

// Check if the queue is full
int isFull(Queue* queue) {
    return queue->size == MAX;
}

// Function to add an element to the queue
void enqueue(Queue* queue, int item) {
    if (isFull(queue)) {
        printf("Queue is full. Cannot enqueue %d.\n", item);
        return;
    }
    queue->rear = (queue->rear + 1) % MAX; // Circular increment
    queue->arr[queue->rear] = item;
    queue->size++;
    printf("%d enqueued to queue.\n", item);
}

// Function to remove and return the front element of the queue
int dequeue(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty. Cannot dequeue.\n");
        return INT_MIN; // Return a sentinel value
    }
    int item = queue->arr[queue->front];
    queue->front = (queue->front + 1) % MAX; // Circular increment
    queue->size--;
    return item;
}

// Function to get the front element of the queue
int front(Queue* queue) {
    if (isEmpty(queue)) {
        printf("Queue is empty.\n");
        return INT_MIN; // Return a sentinel value
    }
    return queue->arr[queue->front];
}
```

```
// Main function to demonstrate queue operations
int main() {
    Queue* queue = createQueue();

    enqueue(queue, 10);
    enqueue(queue, 20);
    enqueue(queue, 30);

    printf("Front element: %d\n", front(queue));

    printf("%d dequeued from queue.\n", dequeue(queue));
    printf("%d dequeued from queue.\n", dequeue(queue));

    enqueue(queue, 40);
    printf("Front element: %d\n", front(queue));

    while (!isEmpty(queue)) {
        printf("%d dequeued from queue.\n", dequeue(queue));
    }

    return 0;
}
```

OUTPUT-

```
10 enqueued to queue.
20 enqueued to queue.
30 enqueued to queue.
Front element: 10
10 dequeued from queue.
20 dequeued from queue.
40 enqueued to queue.
Front element: 30
30 dequeued from queue.
40 dequeued from queue.
```

CONCLUSION-

In this way we understtod the concept of simple queue