Aim: Implement Linear Queue ADT using an array.

Theory:

A queue is a linear data structure that operates on a First-In-First-Out (FIFO) basis, meaning the first element added is the first to be removed. Queue operations typically include:

1. Enqueue - adding an element to the rear.

2. Dequeue - removing an element from the front.

3.Front - retrieving the front element without removing it.

4.Rear- accessing the last element without removing it.

An array implementation of a queue involves using an array to store queue elements. It keeps track of the front and rear indices, where front points to the first element and rear points to the last. This setup allows fixed-sized queues, limited by the array's capacity.

Procedure:

1. Initialize Queue:

   - Define an array of fixed size, capacity.

   - Set front and rear pointers. Typically, front = -1 and rear = -1 to indicate an empty queue.

2. Enqueue (Add an element):

   - Check if the queue is full (rear == capacity - 1).

   - If not full:

   - If the queue is empty (front == -1), set front = 0.

   - Increment rear and place the new element at queue[rear].

3. Dequeue (Remove an element):

   - Check if the queue is empty (front == -1 or front > rear).

   - If not empty:

   - Retrieve the element at queue[front].

   - Increment front to point to the next element.

- If front exceeds rear after removal, reset front and rear to -1, indicating the queue is now empty.

4. Front and Rear Access:

   - For Front, return the element at queue[front] if the queue is not empty.

   - For Rear, return the element at queue[rear] if the queue is not empty.

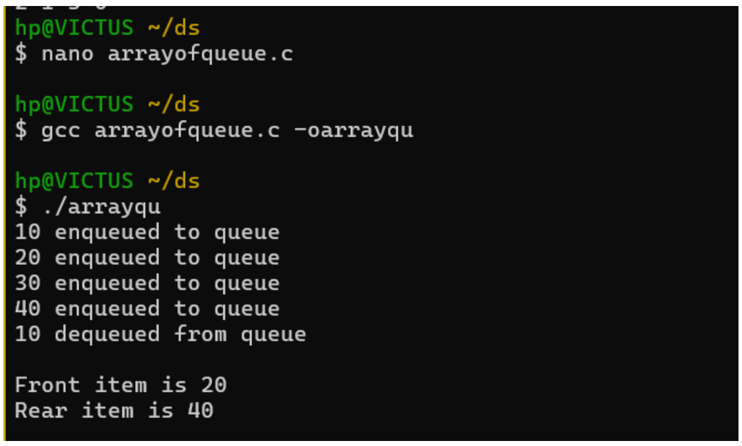
Code:

```
// C program for array implementation of queue
 #include <limits.h>
 #include <stdio.h>
 #include <stdlib.h>
 // A structure to represent a queue
 struct Queue {
 int front, rear, size;
 unsigned capacity;
 int* array;
 };
 // function to create a queue
 // of given capacity.
// It initializes size of queue as 0
 struct Queue* createQueue(unsigned capacity)
 {
 struct Queue* queue = (struct Queue*)malloc(
 sizeof(struct Queue));
 queue->capacity = capacity;
 queue->front = queue->size = 0;
 // This is important, see the enqueue
```

```
queue->rear = capacity- 1;
queue->array = (int*)malloc(
queue->capacity * sizeof(int));
return queue;
}
// Queue is full when size becomes
// equal to the capacity
int isFull(struct Queue* queue)
{
return (queue->size == queue->capacity);
}
// Queue is empty when size is 0
int isEmpty(struct Queue* queue)
{
return (queue->size == 0);
}
// Function to add an item to the queue.
// It changes rear and size
void enqueue(struct 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;
```

```
 printf("%d enqueued to queue\n", item);
 }
 // Function to remove an item from queue.
 // It changes front and size
int dequeue(struct Queue* queue)
 {
 if (isEmpty(queue))
 return INT_MIN;
 int item = queue->array[queue->front];
 queue->front = (queue->front + 1)
 % queue->capacity;
 queue->size = queue->size- 1;
 return item;
 }
 // Function to get front of queue
 int front(struct Queue* queue)
 {
 if (isEmpty(queue))
 return INT_MIN;
 return queue->array[queue->front];
 }
 // Function to get rear of queue
 int rear(struct Queue* queue)
 {
 if (isEmpty(queue))
 return INT_MIN;
 return queue->array[queue->rear];
 }
 // Driver program to test above functions./
 int main()
 {
 struct Queue* queue = createQueue(1000);
 enqueue(queue, 10);
 enqueue(queue, 20);
 enqueue(queue, 30);
 enqueue(queue, 40);
 printf("%d dequeued from queue\n\n",
 dequeue(queue));
 printf("Front item is %d\n", front(queue));
 printf("Rear item is %d\n", rear(queue));
return 0;
 }
```

Output:



```
hp@VICTUS ~/ds
$ nano arrayofqueue.c

hp@VICTUS ~/ds
$ gcc arrayofqueue.c -oarrayqu

hp@VICTUS ~/ds
$ ./arrayqu
10 enqueued to queue
20 enqueued to queue
30 enqueued to queue
40 enqueued to queue
10 dequeued from queue

Front item is 20
Rear item is 40
```

Conclusion: This experiment delves into implementing a Linear Queue ADT using an array. It provides insights into how queues work, including enqueue, dequeue, and front operations, and the utilization of arrays for first-in-first-out (FIFO) data structures.