**ASSIGNMENT – 19**

**1.PROBLEM STATEMENT**

Write a program in C to implement circular queue using array

**2.ALGORITHMS**

Algorithm **Insert\_In\_CircularQueue**

**Input:** The pointer to the array ‘arr’ representing the circular queue, the ‘front’ and ‘rear’ pointers pointing to the element at the start and element at the end of the queue respectively, the element ‘item’ to be inserted into the queue and the size ‘max’ of the queue.

**Output:** The element ‘item’ inserted into the queue at ‘rear’.

**Remarks:** ‘front’ and ‘rear’ must be passed as pointers. An indexing formula x=(x+1) **MOD** max is used to traverse the queue circularly.

**Steps:**

1. **If**(front=rear=0) **then** //when inserting the very first element
2. front=rear=1 //set front and rear to the start of the list
3. arr[rear]=item store item at ‘rear’ position in queue
4. **Return**
5. **Else**
6. rear=(rear+1) **MOD** max //traverse rear using this indexing formula
7. arr[rear]=item //store item at ‘rear’ position in queue
8. **EndIf**
9. **Stop**

Algorithm **Remove\_From\_CircularQueue**

**Input:** The pointer to the array ‘arr’ representing the circular queue, the ‘front’ and ‘rear’ pointers pointing to the element at the start and element at the end of the queue respectively and the size ‘max’ of the queue.

**Output:** The element at front removed from the queue and returned

**Remarks:** ‘front’ and ‘rear’ must be passed as pointers. An indexing formula x=(x+1) **MOD** max is used to traverse the queue circularly.

**Steps:**

1. item=arr[front] //store the item at ‘front’ in item
2. **If**(front=rear) **then** //if front and item are pointing to same location
3. front=rear=0 //set queue to empty state
4. **Return** item
5. **Else**
6. front=(front+1) **MOD** max//traverse front with this indexing formula
7. **EndIf**
8. **Return** item
9. **Stop**

**3.SOURCE CODE**

#include<stdio.h>

#include<stdlib.h>

//function to insert an element in the queue

void enqueue(int\*arr,int \*front,int\*rear,int item,int max)

{

if(\*front==-1&&\*rear==-1) //for inserting first element

{

\*front=\*rear=0; //set front and rear to the start of queue

arr[\*rear]=item; //store item at position of rear in queue

return;

}

else

{

\*rear=(\*rear+1)%max;//traverse rear with this indexing formula

arr[\*rear]=item; //store item at rear in queue

}

}

//function to remove an element from the queue

int dequeue(int\*arr,int\*front,int\*rear,int max)

{

int item;

item=arr[\*front]; //store the element at front in item

if(\*front==\*rear) //if front and rear point to same location

{

\*front=\*rear=-1; //set queue to empty state

return item;

}

else

\*front=(\*front+1)%max;//traverse front with indexing formula

return item;

}

//function to display a queue

void dispq(int\*arr,int\*front,int\*rear,int max)

{

int i;

i=\*front;

if(\*front==-1&&\*rear==-1) // if queue is in empty state

{

printf("Queue is empty");

return;

}

else

{

printf("QUEUE:");

while(i!=\*rear) //traverse i to rear

{

printf("%d ",arr[i]);

i=(i+1)%max; //increment I using indexing formula

}

printf("%d ",arr[\*rear]);

}

}

int main(void)

{

int\*arr,max,front,rear,ch,item;

printf("Enter the size of the queue :");

scanf("%d",&max);

arr=(int\*)calloc(max,sizeof(int));

front=rear=-1;

while(1)

{

printf("\nMENU\n");

printf("1.ENQUEUE\n2.DEQUEUE\n3.DISPLAY\n4.EXIT\n");

printf("Enter your choice: ");

scanf("%d",&ch);

switch(ch)

{

case 1:

if((rear+1)%max==front) //if rear precedes front

{

printf("QUEUE IS FULL");

break;

}

else

{

printf("Enter the element to be inserted: ");

scanf("%d",&item);

enqueue(arr,&front,&rear,item,max);

printf("INSERTED:%d\n",item);

}

dispq(arr,&front,&rear,max);

break;

case 2:

if(front==-1&&rear==-1) //if queue is empty

{

printf("Queue is already empty");

break;

}

else{

item=dequeue(arr,&front,&rear,max);

printf("DELETED:%d\n",item);

}

dispq(arr,&front,&rear,max);

break;

case 3:

dispq(arr,&front,&rear,max);

break;

case 4:

exit(0);

default:

printf("Wrong Choice....Please Try Again");

}

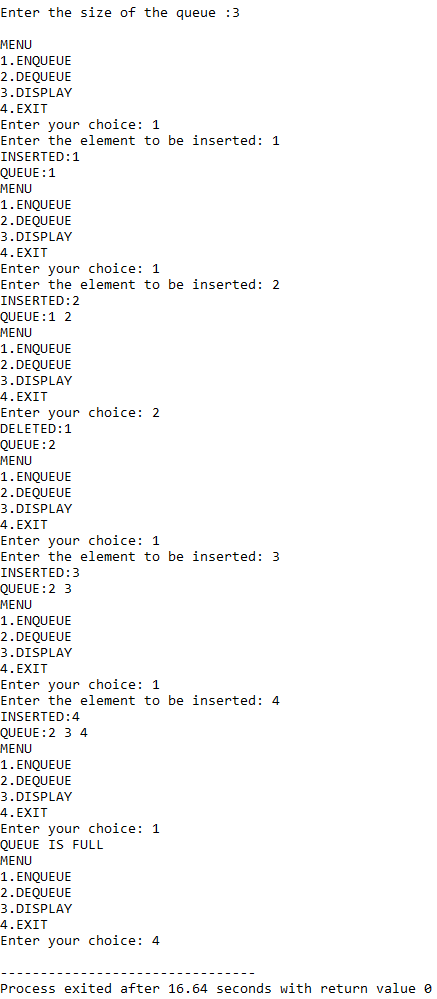
}

return 0;

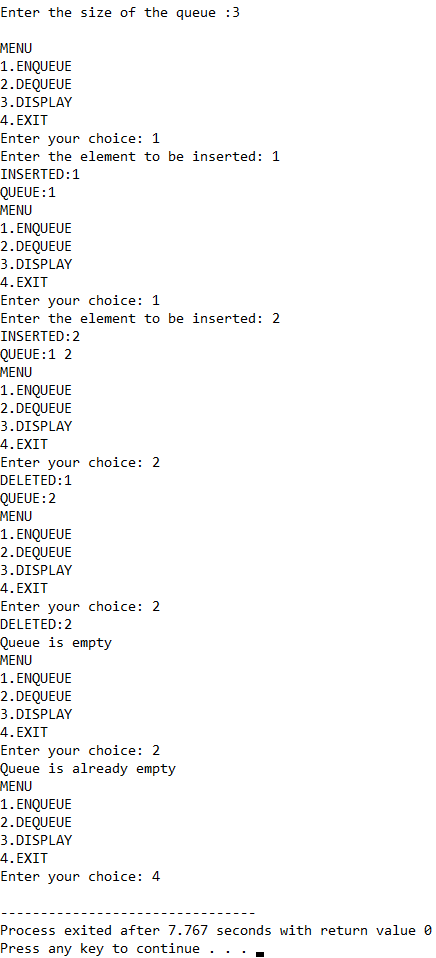
}

**4.OUTPUT**

**SET 1:** Circular insertion in circular queue till queue full

****

**SET 2:** Deletion till queue empty



**5.DISCUSSIONS**

**Variable Description**

* **\*arr:** pointer to hold an array that represents the circular queue.
* **max:** size of the queue.
* **front:** pointer to the element at start in a queue.
* **rear:** pointer to the element at end in a queue.
* **ch:** variable to take user’s input in switch-case-default.
* **item:** variable to store element to be inserted in queue.
* **i:** loop counter.

**Limitations**

* An array has been used to represent the circular queue, since arrays are static data structures, their size cannot be manipulated once they are constructed in the memory, thus they offer less flexibility in size.
* When an element is removed from the queue, it does not get physically erased from the memory rather, it is only removed from the scope of the queue, thus this also leads to inefficient use of memory.

**Uses**

* Circular queues are more memory efficient as compared to linear queues, since in circular queues, if the ‘rear’ pointer reaches the end of the queue, it will check if any space is available at the beginning of the list, and it there is , then the ‘rear’ pointer jumps to the beginning of the queue to utilize the available memory and does not throw a “queue full” message. This is possible due to the mapping formula rear=(rear+1) **MOD** max.

**Future Scope**

* The array representation can be replaced with a linkedlist representation, ending the dependency of the program on contiguous memory locations. It will also allow more flexibility in manipulation of the size.