Experiment No.5
Implement Circular Queue ADT using array
Name: Aryan Gaikwad
Roll No: 09
Date of Performance:
Date of Submission:
Marks:
Sign:

Experiment No. 5: Circular Queue

Aim: To Implement Circular Queue ADT using array

Objective:

Circular Queues offer a quick and clean way to store FIFO data with a maximum size

Theory:

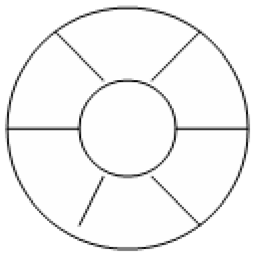
Circular queue is an data structure in which insertion and deletion occurs at an two ends rear and front respectively. Eliminating the disadvantage of linear queue that even though there is a vacant slots in array it throws full queue exception when rear reaches last element. Here in an circular queue if the array has space it never throws an full queue exception. This feature needs an extra variable count to keep track of the number of insertion and deletion in the queue to check whether the queue is full or not.Hence circular queue has better space utilization as compared to linear queue. Figure below shows the representation of linear and circular queue.

Linear queue

Front rear

0	…	…		n

Circular Queue



Algorithm

Algorithm : ENQUEUE(Item)

Input : An item is an element to be inserted in a circular queue.

Output : Circular queue with an item inserted in it if the queue has an empty slot.

Data Structure : Q be an array representation of a circular queue with front and rear pointing to the first and last element respectively.

```
1. If front = 0
       front = 1
       rear =1
       Q[front] = item
2. else
       next=(rear mod length)
       if next!=front then
           rear = next
           Q[rear] = item
       Else
           Print “Queue is full”
       End if
   End if
```

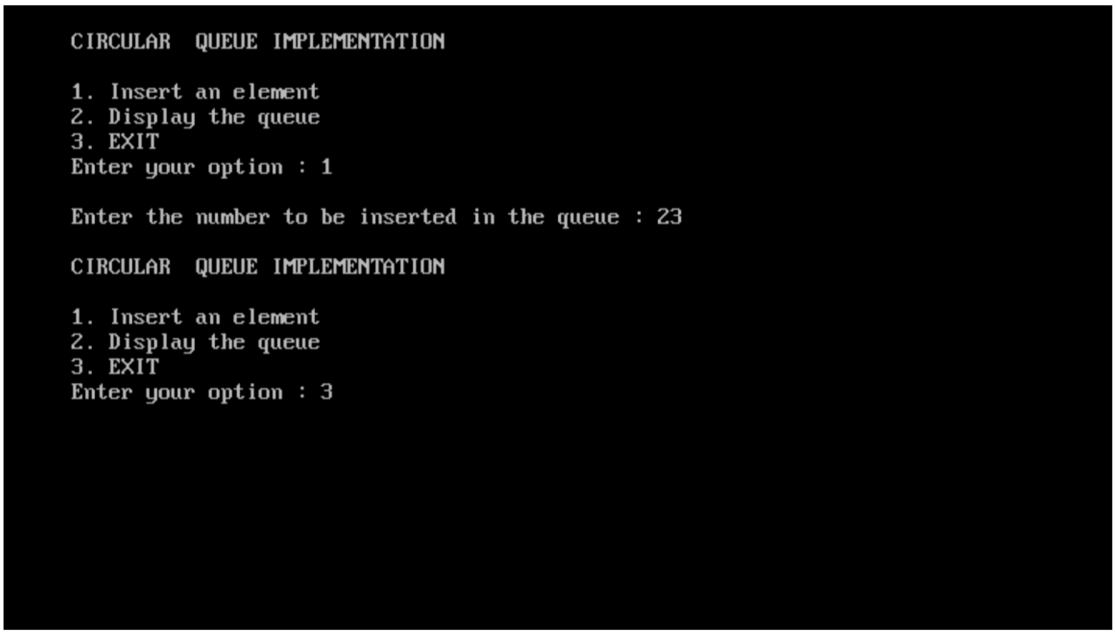
```
    3. stop
Algorithm : DEQUEUE()
Input : A circular queue with elements.
Output :Deleted element saved in Item.
Data Structure : Q be an array representation of a circular queue with front and rear
pointing to the first and last element respectively.
    1. If front = 0
           Print "Queue is empty"
           Exit
    2. else
           item = Q[front]
           if front = rear then
               rear = 0
               front=0
           else
               front = front+1
           end if
       end if
    3. stop
```

Code:

```
#include <stdio.h>

#include <conio.h>

#define MAX 10

int queue[MAX];
int front=-1, rear=-1;

void insert(void);


void display(void);

int main()

{

int option;

clrscr();

do

{

 printf("\n CIRCULAR  QUEUE IMPLEMENTATION ");

 printf("\n");

 printf("\n 1. Insert an element");

 printf("\n 2. Display the queue");

 printf("\n 3. EXIT");

 printf("\n Enter your option : ");

 scanf("%d", &option);

switch(option)

 {

 case 1:

 insert();

break;

case 2:

 display();

break;
```

```
 }
}while(option!=3);
getch();
return 0;
}
void insert()
{
int num;
printf("\n Enter the number to be inserted in the queue : ");
scanf("%d", &num);
if(front==0 && rear==MAX-1)
 printf("\n OVERFLOW");
else if(front==-1 && rear==-1)
{
front=rear=0;
 queue[rear]=num;
}
else if(rear==MAX-1 && front!=0)
{
rear=0;
 queue[rear]=num;
}
else
{
 rear++;
 queue[rear]=num;
}
}
void display()
{
int i;
printf("\n");
if (front ==-1 && rear==-1)
 printf ("\n QUEUE IS EMPTY");
else
{
 if(front<rear)
 {
 for(i=front;i<=rear;i++)
 printf("\t %d", queue[i]);
 }
 else
 {
 for(i=front;i<MAX;i++)
 printf("\t %d", queue[i]);
for(i=0;i<=rear;i++)
 printf("\t %d", queue[i]);
 }
}
}
```

Output:



```
CIRCULAR  QUEUE IMPLEMENTATION

1. Insert an element
2. Display the queue
3. EXIT
Enter your option : 1

Enter the number to be inserted in the queue : 23

CIRCULAR  QUEUE IMPLEMENTATION

1. Insert an element
2. Display the queue
3. EXIT
Enter your option : 3
```

Conclusion:

Q. Explain how Josephus Problem is resolved using circular queue and elaborate on operation used for the same.

The Josephus Problem is a classic problem with an interesting history. It is named after the Jewish historian Flavius Josephus, who, according to legend, and his 40 soldiers were trapped in a cave during a Jewish-Roman War. Rather than surrender to the Romans, they chose to commit suicide. They formed a circle and every third soldier killed the soldier to their left until only one person was left. Josephus, not wanting to die, wanted to position himself in the circle to be the last survivor. He calculated the optimal position.

Here's an elaboration of the steps you provided:

1. Initialize the Circular Queue:

- You create a circular queue of size N, which is essentially an array or data structure that allows you to add and remove elements in a circular manner.

- Fill the circular queue with values from 1 to N, representing the positions of the people in the circle. For example, in a circle of 5 people, the queue might be initially [1, 2, 3, 4, 5].

2. Start the Elimination Process:

- You initialize a variable count to 0, which will keep track of the number of people eliminated.

- You use a while loop to continue the elimination process until only one person is left in the queue (i.e., the size of the queue is greater than 1).

- In each iteration of the loop:

  - Dequeue (remove) the person at the front of the circular queue. This person represents the current position to be checked for elimination.

  - Increment the count by 1 to keep track of how many people you've counted so far.

  - If count is equal to M (the person to be eliminated), you remove the person from the circle by not enqueueing them back into the circular queue. In other words, they are eliminated.

  - If count is not equal to M, you enqueue (add) the person back into the circular queue, which simulates them keeping their position in the circle.

  - You reset the count to 0 after eliminating someone.

3. The Last Person Standing:

- After all eliminations have taken place, there will be only one person left in the circular queue. This person's position is the solution to the Josephus Problem.

The circular queue efficiently models the circular nature of the problem, ensuring that the counting and elimination process wraps around the circle correctly. The time complexity of this solution is O(N*M), where N is the number of people and M is the counting interval. However, there are more efficient algorithms to solve the Josephus Problem with a time complexity of O(N*logN) by directly calculating the position of the last person standing without simulating the eliminations.