NAME :- Manish Shashikant Jadhav
UID :-
BRANCH :- Comps -B. **BRANCH:** B.
EXPERIMENT 2: Implement of given problem statement using Queue.
SUBJECT :- DS (DATA STRUCTURES)

TOPIC 1 :- Implementation of Circular Queue using array.
CODE :-

```c
/*
 * File: circular_queue.c
 * Author: Manish Jadhav
 * Email: manishsj289@gmail.com
 * Created: September 16, 2023
 * Description: This program implements a Queue ADT with a circular array
 */
#include <stdio.h>
#include <stdlib.h>

struct Queue
{
    int front;
    int rear;
    int size;
    char *array;
};

// 1 -> Initialize
struct Queue *initialize_queue(struct Queue *queue, int size)
{
    queue->size = size;
    queue->front = queue->rear = 0;
    queue->array = (char *)malloc(size * sizeof(char));
}

// 2 -> isEmpty
int isEmpty(struct Queue *queue)
{
    if (queue->rear == queue->front)
    {
        return 1;
    }
    return 0;
}
```

```c
// 3 -> isFull
int isFull(struct Queue *queue)
{
    if ((queue->rear + 1) % queue->size == queue->front)
    {
        return 1;
    }
    return 0;
}

// 4 -> enqueue
void enqueue(struct Queue *queue, char item)
{
    if (isFull(queue))
    {
        printf("Queue Overflow\n");
    }
    else
    {
        queue->rear = (queue->rear + 1) % queue->size;
        queue->array[queue->rear] = item;
        printf("Enqueued element: %c\n", item);
    }
}

// 5 -> dequeue
char dequeue(struct Queue *queue)
{
    char val = '\0';
    if (isEmpty(queue))
    {
        printf("Empty queue\n");
    }
    else
    {
        queue->front = (queue->front + 1) % queue->size;
        val = queue->array[queue->front];
    }
    return val;
}
```

```c
// 6 -> front
char front(struct Queue* queue){
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return '\0';
    }
      printf("The front is : %c\n", queue->array[(queue->front + 1) %
queue->size]);
}

// 7 -> rear
char rear(struct Queue* queue){
    if (isEmpty(queue)) {
        printf("Queue is empty\n");
        return '\0';
    }
    printf("The rear is : %c\n", queue->array[queue->rear]);
}

void display(struct Queue *queue)
{
    if (isEmpty(queue))
    {
        printf("Queue is empty\n");
        return;
    }

    printf("Queue elements: ");
    int i = queue->front;
    do
    {
        i = (i + 1) % queue->size;
        printf("%c ", queue->array[i]);
    } while (i != queue->rear);

    printf("\n");
}

int main()
{
    struct Queue q;
    initialize_queue(&q, 5);
```

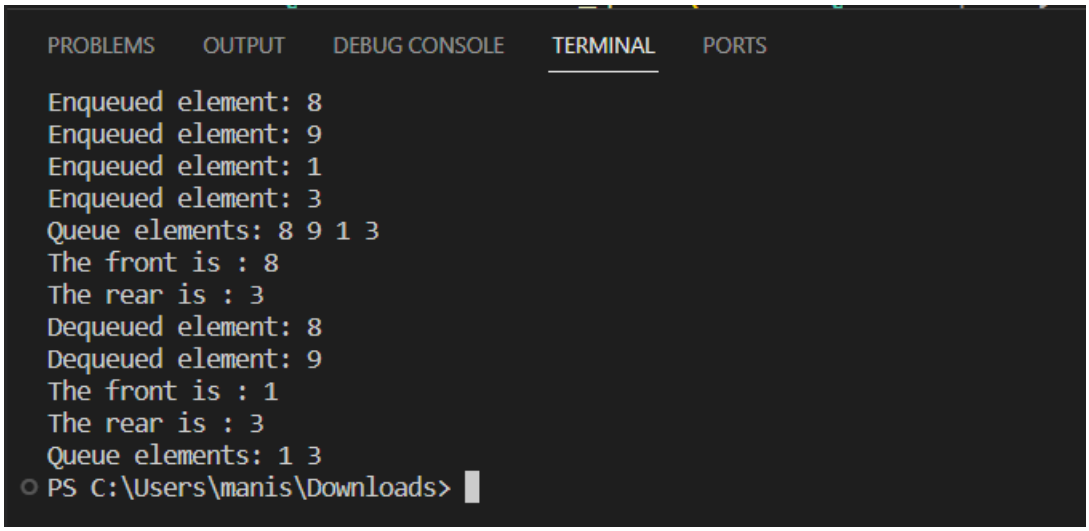
```
    enqueue(&q, '8');
    enqueue(&q, '9');
    enqueue(&q, '1');
    enqueue(&q, '3');


    display(&q);
    front(&q);
    rear(&q);
    printf("Dequeued element: %c\n", dequeue(&q));
    printf("Dequeued element: %c\n", dequeue(&q));


    front(&q);
    rear(&q);
    display(&q);

    return 0;
}
```

OUTPUT :-



Algorithm:

1. Initialize Queue

- Create a structure called `Queue` with fields for front, rear, size, and an array to hold elements.

- Implement an `initialize_queue` function that takes a pointer to a `Queue` structure and a size as input.

- Set the size of the queue and initialize both front and rear to 0.

- Allocate memory for the array using `malloc`.

2. isEmpty

- Implement an `isEmpty` function that takes a pointer to a `Queue` structure.

- Check if the rear and front of the queue are at the same index; if they are, the queue is empty and return 1; otherwise, return 0.

3. isFull

- Implement an `isFull` function that takes a pointer to a `Queue` structure.

- Check if the next position after the rear is equal to the front; if it is, the queue is full and return 1; otherwise, return 0.

4. Enqueue

- Implement an `enqueue` function that takes a pointer to a `Queue` structure and an element to be added.

- Check if the queue is full using the `isFull` function. If it's full, print "Queue Overflow."

- Otherwise, increment the rear pointer to the next position, and store the element at that position in the array.

5. Dequeue

- Implement a `dequeue` function that takes a pointer to a `Queue` structure.

- Check if the queue is empty using the `isEmpty` function. If it's empty, print "Empty queue" and return a default character (e.g., '\0').

- Otherwise, increment the front pointer to the next position and return the element at that position in the array.

6. Front

- Implement a `front` function that takes a pointer to a `Queue` structure.

- Check if the queue is empty using the `isEmpty` function. If it's empty, print "Queue is empty" and return a default character (e.g., '\0').

- Otherwise, print the element at the next position after the front in the array.

7.Rear

- Implement a `rear` function that takes a pointer to a `Queue` structure.

- Check if the queue is empty using the `isEmpty` function. If it's empty, print "Queue is empty" and return a default character (e.g., '\0').

- Otherwise, print the element at the rear position in the array

8. Display Queue

- Implement a `display` function that takes a pointer to a `Queue` structure.

- Check if the queue is empty using the `isEmpty` function. If it's empty, print "Queue is empty."

- Otherwise, loop through the elements in the queue, starting from the next position after the front and ending at the rear position, and print each element.

9. Main Function

- In the `main` function:

  - Declare a `Queue` structure variable `q`.

  - Initialize the queue using the `initialize_queue` function with a size of 5.

  - Enqueue several elements into the queue.

  - Display the queue.

  - Retrieve and print the front and rear elements.

  - Dequeue two elements and print them.

  - Display the updated queue.

TOPIC 2 :- N-Series and Chill
CODE :-

```c
/*
 * File: nqueues.c
 * Author: Manish Jadhav
 * Email: manishsj289@gmail.com
 * Created: September 16, 2023
 * Description: This program implements an n-series queues data structure
 */

/*
 N-Series and chill
*/

#include "queue.c"
#include <stdio.h>
#include <stdlib.h>

struct NQueues
{
    int *front;
    int *rear;
    unsigned total_queues;
    unsigned size_per_queue;
    struct Queue *array[100];
};

struct NQueues *createNQueues(int n, int capacity_per_queue)
{
    struct NQueues *nqueues = (struct NQueues *)malloc(sizeof(struct
NQueues));
    nqueues->total_queues = n;
    nqueues->size_per_queue = capacity_per_queue;
    for (int i = 0; i < n; i++)
    {
        nqueues->array[i] = initialize_queue(capacity_per_queue);
    }
    return nqueues;
}

void add_episode(struct NQueues *nqueues, int episode_id, int
queue_number)
```

```
{
    enqueue(nqueues->array[queue_number], episode_id);
}

char watch_next_episode(struct NQueues *nqueues, int queue_number)
{
    char ep = dequeue(nqueues->array[queue_number]);
    return ep;
}

void display_queue(struct NQueues *nqueues, int queue_number)
{
    display(nqueues->array[queue_number]);
}

void display_all(struct NQueues *nqueues)
{
    for (int i = 0; i < nqueues->total_queues; i++)
    {
        printf("> Queue No: %d", i + 1);
        display(nqueues->array[i]);
    }
}

int main()
{
    struct NQueues *nqueues = createNQueues(3, 5);
    add_episode(nqueues, 77, 0);
    add_episode(nqueues, 65, 0);
    add_episode(nqueues, 78, 0);
    add_episode(nqueues, 73, 1);
    add_episode(nqueues, 83, 1);
    add_episode(nqueues, 72, 1);
    add_episode(nqueues, 80, 2);
    add_episode(nqueues, 85, 2);
    add_episode(nqueues, 82, 2);

    display_all(nqueues);

    char next_episode = watch_next_episode(nqueues, 0);
    printf("Watched episode: %d : %c\n", next_episode, next_episode);
```

```
    printf("Queue after watching episode:\n");
    display_queue(nqueues, 0);

    next_episode = watch_next_episode(nqueues, 0);
    next_episode = watch_next_episode(nqueues, 0);

    // Underflow check
    next_episode = watch_next_episode(nqueues, 0);

    // Empty Queue Display
    printf("\nQueue after watching episode:");
    display_queue(nqueues, 0);

    // Free Space
    free(nqueues->array);
    free(nqueues);

   return 0;
}
```

OUTPUT :-

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

                    cd "d:\Manish\DS SPIT\" ; if ($?) { gcc nqueues.c -o nqueues } ; if ($?) { .\nqueues }
> Queue No: 1
> Queue Front
> M
> A
> N
> Queue Rear

> Queue No: 2
> Queue Front
> I
> S
> H
> Queue Rear

> Queue No: 3
> Queue Front
> P
> U
> R
> Queue Rear

Watched episode: 77 : M
Queue after watching episode:

> Queue Front
> A
> N
> Queue Rear


> Queue already Empty!

Queue after watching episode:
> Queue Front
> Queue Rear

PS D:\Manish\DS SPIT>
```

Algorithm:

1. Structure Definition:

- Define a structure `struct NQueues` to store the N queues. Include fields for the front and rear pointers for each queue, the total number of queues, the size per queue, and an array of pointers to `Queue` structures.

2. Initialization:

- Implement a function `createNQueues(int n, int capacity_per_queue)` to create the NQueues data structure. Allocate memory for `struct NQueues`. Initialize the total number of queues and the size per queue.

- Initialize each queue in the array using the `initialize_queue(capacity_per_queue)` function from "queue.c".

3. Adding an Episode:

- Implement a function `add_episode(struct NQueues *nqueues, int episode_id, int queue_number)` to add an episode to a specific queue.

- Use the `enqueue()` function from "queue.c" to add the episode to the specified queue.

4. Watching Next Episode:

- Implement a function `watch_next_episode(struct NQueues *nqueues, int queue_number)` to watch the next episode from a specific queue.

- Use the `dequeue()` function from "queue.c" to remove and return the next episode from the specified queue.

5. Displaying a Queue:

- Implement a function `display_queue(struct NQueues *nqueues, int queue_number)` to display the contents of a specific queue using the `display()` function from "queue.c".

6. Displaying All Queues:

- Implement a function `display_all(struct NQueues *nqueues)` to display all the queues, including their contents.

7. Main Function:

- In the `main()` function:

- Create an instance of `struct NQueues` with the desired number of queues and capacity per queue using `createNQueues()`. Add episodes to various queues using the `add_episode()` function.

- Display all the queues using `display_all()`.

- Watch episodes from a specific queue using `watch_next_episode()`.

- Handle underflow conditions when trying to watch episodes from an empty queue.

- Free allocated memory for the data structure and its queues at the end.

Page No.
Date:
YOUVA

Experiment No. 2

* Aim:- Implement a given problem statement using Queue.

* Theory :-

• Queue:- Queue is a linear data structure that is open at both ends and the operations are ~~per~~ performed in First-In-First-Out (FIFO) order.

In Queue, all additions are made at one end and all deletions are made at another end.

The first entry that will be removed from the queue, is called 'front' of the queue. The position of the last entry in the queue is called the rear of the queue.

• Characteristics of Queue:-

a) Queue can handle multiple data.
b) We can access both ends.
c) They are fast and flexible.

• Advantages of Queue:-

a) A large amount of data can be managed efficiently with use.
b) Useful when service is used by multiple consumers.
c) Fast in speed for data inter-process communication.

- Disadvantage of Queue:-
  a) Limited Space.
  b) Maximum number of queue must be defined prior.
  c) Operations from middle are time consuming.

- What is Circular Queue?
  A circular Queue is an extended version of normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

- Basic Operations on Queue:-
  a) enqueue():- Insert elements at rear end.
  b) dequeue():- Removes element at front end.
  c) front():- Returns first element from queue.
  d) rear():- Returns last element from queue.
  e) isEmpty():- Indicates whether queue is empty or not.
  f) isFull():- Indicates whether queue is full or not.
  g) size():- Returns size of queue.

- Algorithm:-
  a) Create structure for Queue.
  b) Initialize the Queue.
  c) Check if queue is Empty, if yes return true(1) else false (0).
  d) Check is Queue is Full, if yes return true (1), else false (0).
  e) Enqueue elements:-
  If queue is full return "overflow", else ~~in~~ all element and ~~return~~ increment rear index.

f) Dequeue an element :-
If queue is empty, return "Underflow", else return the element and ~~decrement~~ increment front index.

g) Get Rear and Front element.

h) Display the Queue.

* Conclusion :-
Hence, by completing this experiment I came to know about implementation of circular queue.