LAB 6 - (1)Qns - circular_queue

Given the size of the circular queue N and Q queires. Queires are related to inseriton and deletion of the elements in the circular queue. For insertion the querey is "1 'element' ", For deletion querey is "2". 1.Write code for enqueue and dequeue of elements from the list according to the query given. 2.Return the number of prime numberes from the circular queue after completing all the queueries. Given the size of the circular queue N and Q queires. Queires are related to inseriton and deletion of the elements in the circular queue. For insertion the querey is "1 'element' ", For deletion querey is "2". 1.Write code for enqueue and dequeue of elements from the list according to the query given. 2.Return the number of prime numberes from the circular queue after completing all the queueries.

Input Format

1st line has the Number N length of queue 2nd line has number of queries Next Q lines has the quereies. 1st line has the Number N length of queue 2nd line has number of queries Next Q lines has the quereies.

Constraints

N<10 Q<10

Output Format

Prints the number of prime’s after completion of the queries.

Sample Input 0

```
5
5
1 5
1 3
2
1 5
1 2
```

Sample Output 0

```
3
```

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
```

```
#define MAX_SIZE 10

struct CircularQueue {
    int queue[MAX_SIZE];
    int front, rear;
    int size;
};

void initQueue(struct CircularQueue *cq, int size) {
    cq->front = cq->rear = -1;
    cq->size = size;
}

void enqueue(struct CircularQueue *cq, int element) {
    if ((cq->rear + 1) % cq->size == cq->front) {
        // Circular queue is full
        return;
    }
    if (cq->front == -1) {
        // Queue is empty
        cq->front = cq->rear = 0;
        cq->queue[cq->rear] = element;
    } else {
        cq->rear = (cq->rear + 1) % cq->size;
        cq->queue[cq->rear] = element;
    }
}

int dequeue(struct CircularQueue *cq) {
    if (cq->front == -1) {
        // Queue is empty
        return -1;
    }
    int element = cq->queue[cq->front];
    if (cq->front == cq->rear) {
        // Last element in the queue
        cq->front = cq->rear = -1;
    } else {
        cq->front = (cq->front + 1) % cq->size;
    }
    return element;
}

bool isPrime(int n) {
    if (n <= 1) {
        return false;
    }
    for (int i = 2; i <= sqrt(n); i++) {
        if (n % i == 0) {
            return false;
        }
    }
    return true;
}
```

```
int countPrimesInCircularQueue(struct CircularQueue *cq) {
    int count = 0;
    if (cq->front == -1) {
        // Queue is empty
        return count;
    }
    int current = cq->front;
    while (current != cq->rear) {
        if (isPrime(cq->queue[current])) {
            count++;
        }
        current = (current + 1) % cq->size;
    }
    if (isPrime(cq->queue[cq->rear])) {
        count++;
    }
    return count;
}

int main() {
    struct CircularQueue cq;
    int size;
    scanf("%d", &size);
    initQueue(&cq, size);
    int q;
    scanf("%d", &q);
    char query[4];
    int element;

    for (int i = 0; i < q; i++) {
        scanf("%s", query);
        if (query[0] == '1') {
            scanf("%d", &element);
            enqueue(&cq, element);
        } else if (query[0] == '2') {
            dequeue(&cq);
        }
    }

    int primeCount = countPrimesInCircularQueue(&cq);
    printf("%d\n", primeCount);

    return 0;
}
```