General instructions

```
# Linear queue
gcc -Wall -o test_queue test_queue.c Queue.c
./test_queue

# Circular queue
gcc -Wall -o test_cqueue test_cqueue.c CQueue.c
./test_cqueue
```

Note : test code attached below

test_common.h

```
#ifndef TEST_COMMON_H
#define TEST_COMMON_H

#include <stdio.h>
static int tests_run = 0;
static int tests_passed = 0;

#define CHECK(condition, description)
    do {
        tests_run++;
        if (condition) {
            tests_passed++;
            printf("  [PASS] %s\n", description);
        } else {
            printf("  [FAIL] %s  (line %d)\n", description, __LINE__);
        }
    } while (0)

#define SECTION(title) printf("\n--- %s ---\n", title)

#endif
```

test_queue.c

```
#include "Queue.h"
#include "test_common.h"

int main(void) {
    Queue q;

    SECTION("A freshly initialized queue is empty");
    initializeQueue(&q);
    CHECK(isEmpty(&q) == true, "isEmpty() is true right after
initializeQueue()");
    CHECK(isFull(&q) == false, "isFull() is false right after
initializeQueue()");

    SECTION("enqueue()/dequeue() preserve FIFO order");
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    CHECK(dequeue(&q) == 10, "1st dequeue() returns the 1st value
enqueued (10)");
    CHECK(dequeue(&q) == 20, "2nd dequeue() returns the 2nd value
enqueued (20)");
    CHECK(dequeue(&q) == 30, "3rd dequeue() returns the 3rd value
enqueued (30)");
    CHECK(isEmpty(&q) == true, "queue is empty again after dequeuing
everything");

    SECTION("isFull() becomes true after MAX_SIZE enqueues");
    initializeQueue(&q);
    for (int i = 0; i < MAX_SIZE; i++) {
        enqueue(&q, i * 100);
    }
    CHECK(isFull(&q) == true, "isFull() is true after enqueuing
MAX_SIZE items");

    SECTION("enqueue() on a full queue must not corrupt existing
data");
    int frontValueBefore = q.items[q.front];
    enqueue(&q, 999); // should be rejected -- queue is already full
    CHECK(isFull(&q) == true, "queue is still reported full after the
rejected enqueue()");
    CHECK(q.items[q.front] == frontValueBefore, "front item is
unchanged after a rejected enqueue()");

    SECTION("dequeue() on an empty queue must leave state
consistent");
    initializeQueue(&q);
    dequeue(&q); // should be rejected/handled gracefully -- queue
starts empty
    CHECK(isEmpty(&q) == true, "queue is still reported empty after
dequeue() on an empty queue");

    SECTION("LIMITATION: a linear queue cannot reuse freed space
without a full reset");
```

```
    initializeQueue(&q);
    for (int i = 0; i < MAX_SIZE; i++) enqueue(&q, i); // fill:
0,1,2,3,4
    dequeue(&q); // remove 0 -- one logical slot freed
    dequeue(&q); // remove 1 -- a second logical slot freed
    CHECK(isFull(&q) == true,
          "even with 2 of 5 slots logically free, isFull() is STILL
true "
          "(rear never moves backward) -- this is exactly the problem
a Circular Queue solves");

    printf("\n%d / %d checks passed\n", tests_passed, tests_run);
    return (tests_passed == tests_run) ? 0 : 1;
}
```

test_cqueue.c

```
#include "CQueue.h"
#include "test_common.h"

int main(void) {
    CircularQueue q;

    SECTION("A freshly initialized circular queue is empty");
    initQueue(&q);
    CHECK(isEmpty(&q) == true, "isEmpty() is true right after
initQueue()");
    CHECK(isFull(&q) == false, "isFull() is false right after
initQueue()");

    SECTION("enqueue()/dequeue() preserve FIFO order");
    enqueue(&q, 10);
    enqueue(&q, 20);
    enqueue(&q, 30);
    CHECK(dequeue(&q) == 10, "1st dequeue() returns the 1st value
enqueued (10)");
    CHECK(dequeue(&q) == 20, "2nd dequeue() returns the 2nd value
enqueued (20)");
    CHECK(dequeue(&q) == 30, "3rd dequeue() returns the 3rd value
enqueued (30)");
    CHECK(isEmpty(&q) == true, "queue is empty again after dequeuing
everything");

    SECTION("isFull() becomes true after MAX_SIZE enqueues");
    initQueue(&q);
    for (int i = 0; i < MAX_SIZE; i++) {
        enqueue(&q, i * 100);
    }
    CHECK(isFull(&q) == true, "isFull() is true after enqueuing
MAX_SIZE items");

    SECTION("enqueue() on a full queue must not corrupt existing
data");
    int frontValueBefore = q.items[q.front];
    enqueue(&q, 999); // should be rejected -- queue is already full
```

```
    CHECK(isFull(&q) == true, "queue is still reported full after the
rejected enqueue()");
    CHECK(q.items[q.front] == frontValueBefore, "front item is
unchanged after a rejected enqueue()");

    SECTION("dequeue() on an empty queue must leave state
consistent");
    initQueue(&q);
    dequeue(&q); // should be rejected/handled gracefully -- queue
starts empty
    CHECK(isEmpty(&q) == true, "queue is still reported empty after
dequeue() on an empty queue");

    SECTION("ADVANTAGE: a circular queue DOES reuse freed space via
wraparound");
    initQueue(&q);
    for (int i = 0; i < MAX_SIZE; i++) enqueue(&q, i); // fill:
0,1,2,3,4
    dequeue(&q); // remove 0 -- front advances
    dequeue(&q); // remove 1 -- front advances again
    CHECK(isFull(&q) == false,
          "after freeing 2 slots, isFull() correctly becomes false
(unlike a linear queue)");

    enqueue(&q, 100); // should wrap around and reuse the slot value 0
occupied
    enqueue(&q, 200); // should wrap around and reuse the slot value 1
occupied
    CHECK(isFull(&q) == true,
          "re-filling those 2 freed slots correctly brings the queue
back to full -- "
          "this wraparound reuse is exactly what succ() provides");
    CHECK(dequeue(&q) == 2,
          "FIFO order still holds across the wraparound: the oldest
remaining "
          "item (2) comes out first, ahead of the newly wrapped-in 100
and 200");

    printf("\n%d / %d checks passed\n", tests_passed, tests_run);
    return (tests_passed == tests_run) ? 0 : 1;
}
```