Data structure
Workshop 3

I. Exercise

1.1 The main advantage of a circular queue over an ordinary sequential queue is _______.

1.2 The following is the definition and partial operations of a circular queue based on an array. Please complete the codes:

```
#include <stdio.h>
#include <stdlib.h>
#define MAX_SIZE 4    // Maximum capacity of the queue

typedef struct {
    int data[MAX_SIZE];
    int front;    // Front pointer (points to the front element)
    int rear;     // Rear pointer (points to the position next to the last element)
} CirQueue;

// Initialize the queue
void initQueue(CirQueue *q) {
    if (q == NULL) return;
     ____________________(1);
     ____________________(2);
}

// Check if the queue is empty
int isEmpty(CirQueue *q) {
    if (q == NULL) return -1;
    return ___________________(3)_;    // Condition for an empty queue
}


// Check if the queue is full ( sacrifices one unit)
int isFull(CirQueue *q) {
    if (q == NULL) return -1;
    return __________________(4)__;    // Condition for a full queue
}

// Enqueue operation (return 1 if successful, 0 if failed)
int enqueue(CirQueue *q, int val) {
    if (q == NULL || isFull(q)) {
        printf("Enqueue failed: Queue is full\n");
        return 0;
    }
    __________________(5)__;    //Store the element at the rear position
    __________________(6)__;    // Update the rear pointer (circular movement)
    return 1;
}

// Dequeue operation
int dequeue(CirQueue *q) {
    if (q == NULL || isEmpty(q)) {
        printf("Dequeue failed: Queue is empty\n");
        return -1;
    }
    int val = __________________(7)__;    // Get the front element
    _________________(8)___;    // Update the front pointer (circular movement)
    return val;
}
```

II. Experiment

Implement a Stack Using a Linked List and Test It

Implement a stack using a singly linked list, with the following required operations:

1. Initialize the stack (initStack)
2. Push operation (push): Add an element to the top of the stack
3. Pop operation (pop): Remove and return the top element (return -1 with a prompt if the stack is empty)
4. Get the top element (getTop): Return the top element (return -1 with a prompt if the stack is empty)
5. Check if the stack is empty (isEmpty): Return 1 if empty, 0 if not
6. Destroy the stack (destroyStack): Free memory of all nodes

Requirements for Test Code:

After implementing the stack, write test code that must cover the following scenarios to verify all functions:

1. Initialization and empty stack verification

After initializing the stack, immediately call isEmpty to check if it is empty and print the result.

Perform a pop operation on the empty stack, verifying that it prompts "stack is empty" and returns the expected value (e.g., -1).

Perform a getTop operation on the empty stack, verifying that it prompts "stack is empty" and returns the expected value (e.g., -1).

2. Push operation verification

Push at least 3 different integers (e.g., 10, 20, 30) consecutively. After each push, briefly prompt "Push successful: X".

After pushing, call isEmpty to confirm the stack is not empty and print the result.

Call getTop to get the top element, verifying it is the last pushed element (e.g., 30).

3. Pop operation verification

Perform one pop operation, print the popped element (should match the current top, e.g., 30), then call getTop to confirm the top has been updated (e.g., 20).

Perform pop operations for the remaining elements consecutively (e.g., pop 20, 10) until the stack is empty.

After all elements are popped, call isEmpty to confirm the stack is empty and print the result.

Repeated operations and boundary verification

Push 2 new elements again (e.g., 5, 6), then call getTop to verify the top is correct (e.g., 6).

After one pop operation (popping 6), confirm the top is updated to 5.

4. Stack destruction verification

Call destroyStack to destroy the stack. Then attempt to call getTop or pop to verify the stack can no longer operate normally (to avoid wild pointer access).