Smart Bridge Load Monitor

Assignment Question 10 – Mechanical

Name: S Kunaal Jaiswal

Roll Number: ME24I1013

Date of Submission: 15/April/2025

## 1. Problem Statement: What Are We Solving?

This project simulates a Smart Bridge Load Monitor system that tracks and evaluates structural components of a bridge (like Beam, Cable, Deck, Joint, etc.) using data structures. The system ensures structural safety by managing incoming load test requests, storing test results, and making them available for maintenance review. The core idea is to design a basic real-time simulator that uses common Data Structures and Algorithms (DSA) to model a simplified but realistic version of how bridges are monitored in smart cities.

## 2. Objectives

Simulate load testing and logging using C programming.

Apply Queues for managing test requests and Arrays for logging history.

Ensure continuous performance by archiving old results and processing new ones.

Gain hands-on experience in real-time data handling using basic DSA.

## 3. Design Explanation

The system design consists of two main parts:

Load Request System: This uses a queue (First In, First Out) to track incoming sensor-based load tests. Each test is stored and processed in the order it was added.

Load History Unit: This uses a fixed-size circular array to store the last six test results. Once full, the oldest results are automatically removed (archived) to make room for new entries.

The menu-driven program allows users to interact with the system by adding tests, processing them, logging results, and viewing recent history.

## 4. Why These Data Structures?

Queue (Array-based): A Queue is best for handling sensor requests where the first test to come in must be processed first. It models real-time task scheduling efficiently.

Circular Array (Fixed Size): This structure is ideal for logging limited test results. It saves memory, avoids unnecessary data shifting, and automatically archives old data when full.

## 5. Logic of the Code – Step-by-Step Explanation

1. Initialization: Queue and history buffer are initialized to represent an empty state.

2. Add Load Test:

- The user enters a test name (e.g., "Beam").

- This is added to the queue unless it is full.

3. Process Load Test:

- The first test in the queue is removed and displayed.

- If the queue is empty, an appropriate message is shown.

4. Log Load Result:

- The user inputs the result (e.g., "Pass", "Fail").

- This is added to the history. If the history is full, the oldest result is overwritten.

5. Display Load History:

- All stored results in the buffer are printed in the correct order.

6. Exit:

- Ends the program execution.

## 6. Variables and Functions – Explained Clearly

## 7. Code Implementation

```

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_TESTS 5

#define HISTORY\_SIZE 6

typedef struct {

char tests[MAX\_TESTS][20];

int front, rear;

} Queue;

void enqueue(Queue \*q, char \*test) {

if (q->rear == MAX\_TESTS - 1) {

printf("Queue is full!\n");

return;

}

q->rear++;

strcpy(q->tests[q->rear], test);

if (q->front == -1) {

q->front = 0;

}

}

void dequeue(Queue \*q) {

if (q->front == -1) {

printf("Queue is empty!\n");

return;

}

printf("Processing test: %s\n", q->tests[q->front]);

if (q->front == q->rear) {

q->front = q->rear = -1;

} else {

q->front++;

}

}

typedef struct {

char results[HISTORY\_SIZE][20];

int start, count;

} LoadHistory;

void logResult(LoadHistory \*lh, char \*result) {

int pos = (lh->start + lh->count) % HISTORY\_SIZE;

strcpy(lh->results[pos], result);

if (lh->count < HISTORY\_SIZE) {

lh->count++;

} else {

lh->start = (lh->start + 1) % HISTORY\_SIZE;

}

}

void displayHistory(LoadHistory \*lh) {

printf("Load History:\n");

for (int i = 0; i < lh->count; i++) {

printf("%s\n", lh->results[(lh->start + i) % HISTORY\_SIZE]);

}

}

int main() {

Queue q;

q.front = -1;

q.rear = -1;

LoadHistory lh;

lh.start = 0;

lh.count = 0;

char test[20], result[20];

int choice;

while (1) {

printf("\nMenu:\n");

printf("1. Add Load Test\n");

printf("2. Process Load Test\n");

printf("3. Log Load Result\n");

printf("4. Show Load History\n");

printf("5. Exit\n");

printf("Enter choice: ");

scanf("%d", &choice);

while (getchar() != '\n');

switch (choice) {

case 1:

printf("Enter test name: ");

fgets(test, 20, stdin);

test[strcspn(test, "\n")] = 0;

enqueue(&q, test);

break;

case 2:

dequeue(&q);

break;

case 3:

printf("Enter test result: ");

fgets(result, 20, stdin);

result[strcspn(result, "\n")] = 0;

logResult(&lh, result);

break;

case 4:

displayHistory(&lh);

break;

case 5:

exit(0);

default:

printf("Invalid choice! Try again.\n");

}

}

return 0;

}

```

## 8. Sample Output (Screenshot or Console Print)

Menu:

1. Add Load Test

2. Process Load Test

3. Log Load Result

4. Show Load History

5. Exit

Enter choice: 1

Enter test name: Beam

Enter choice: 3

Enter test result: Passed

Enter choice: 4

Load History:

Passed

## 9. Viva Preparation – What to Expect

Q1: Why did you use a queue for load tests?

Because sensor data or test requests arrive over time and must be processed in the same order they come in (FIFO).

Q2: Why circular array for history?

It allows us to store only the most recent test results and replaces old data automatically without shifting all array elements.

Q3: Time Complexities?

- Enqueue/Dequeue: O(1)

- Logging Result: O(1)

- Displaying History: O(n) where n ≤ 6

Q4: Edge Cases?

- Queue full and empty conditions handled.

- Circular buffer overwriting logic for full history.

- Input buffer clearing to prevent input skipping.