DSA Project Report

Title: Hydroelectric Dam Flow Manager

Name: Harshit Tiwari

Roll Number: Me24b1051

Date: 15/04/2025

1. Problem Statement

The objective of this project is to design a Hydroelectric Dam Flow Manager, which simulates and controls water flow using different data structures. It manages:
- Water flow requests via sensors.
- Emergency response to critical adjustments.
- Logging power output.
- Tracking component maintenance.
- Tuning high-priority components.

This project integrates arrays, stack, queue, singly linked list, doubly linked list, and circular linked list to model each real-world behavior appropriately.

2. Project Objectives

- Handle flow requests using a queue (FIFO).
- Deal with emergency adjustments using a stack (LIFO).
- Log power output using a fixed-size array.
- Track worn and repaired components using singly and doubly linked lists.
- Maintain a circular loop of high-priority components using a circular linked list.
- Simulate real-time water and component management effectively.

3. Design Explanation

Component	Data Structure	Reason for Choice
Flow Requests	Queue	First come, first served order (FIFO).

Emergency Adjustments	Stack	Most recent first handled (LIFO).
Power Output Log	Array	Simple log with fixed slots.
Worn Component Tracking	Singly Linked List	Easy to add/delete worn parts in one direction.
Repaired Component Review	Doubly Linked List	Can traverse forward/backward for inspection.
Priority Tuning	Circular Linked List	Constant looping through urgent components.

4. Logic of the Code (Step-by-Step)

Task A – Flow Request + Emergency Adjustments
- enqueue() stores 6 flow inputs.
- dequeue() removes from the queue and push() adds to a stack.
- pop() is used to show emergency order (last-in, first-out).

Task B – Power Output Log
- 7 power outputs are entered one by one.
- After the 5th output, oldest one is transmitted (deleted), and new one is added.
- Uses basic array shifting logic.

Task C – Maintenance Tracker
- insert_worn() adds "Turbine" and "Pump" to singly linked list.
- "Turbine" is deleted and moved into a doubly linked list for repair.
- traverse_repaired_forward() and traverse_repaired_backward() simulate review.

Task D – Priority Tuning
- "Gate" and "Spillway" are added into a circular list.
- We loop through them multiple times to simulate constant priority monitoring.

5. Variables and Functions Used

Variable / Function	Purpose
queue, enqueue(), dequeue()	For flow requests.

stack, push(), pop()	For emergency handling.
power_log[], log_power()	Logs and manages power output.
SNode, insert_worn(), delete_worn()	Tracks worn components.
DNode, insert_repaired()	Tracks repaired components.
traverse_repaired_forward()	View repaired list forward.
traverse_repaired_backward()	View repaired list backward.
CNode, insert_circular()	Inserts urgent tuning components.
traverse_circular()	Loops through priority tuning list.

6. Sample Output

```
--- Task A: Flow and Emergency ---
Emergency Adjustment Order (LIFO):
Valve
Pump
Reservoir
Spillway
Gate
Turbine

--- Task B: Power Output Log ---
Transmitting (Oldest): Pow1
Transmitting (Oldest): Pow2
Current Power Log:
Pow3
Pow4
Pow5
Pow6
Pow7

--- Task C: Worn Component Tracker ---
Repaired Components (Forward):
Turbine
Repaired Components (Backward):
Turbine

--- Task D: Priority Tuning ---
```

```
Circular Priority Tuning:
Gate
Spillway
Gate
Spillway
```

7. CODE

```
//me24b1051
//harshit tiwari
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// We define some limits for arrays and strings
#define MAX 6
#define SIZE 20
#define POWER_LOG_SIZE 5

// ------------------------
// Task A: Queue + Stack
// ------------------------

char* flow_requests[] = {"Turbine", "Gate", "Spillway", "Reservoir", "Pump", "Valve"};

char queue[MAX][SIZE]; // To hold flow requests
int front = -1, rear = -1;

char stack[MAX][SIZE]; // To hold emergency flow adjustments
int top = -1;

// Adds a request to the queue
void enqueue(char* request) {
    if (rear == MAX - 1) {
        return; // queue is full
    }
    if (front == -1) front = 0;
    rear++;
    strcpy(queue[rear], request);
}

// Removes a request from the queue
char* dequeue() {
    if (front == -1 || front > rear) {
        return NULL; // queue is empty
    }
    return queue[front++];
}

// Pushes a request onto the stack
void push(char* request) {
    if (top == MAX - 1) return; // stack is full
    top++;
    strcpy(stack[top], request);
}

// Pops the top of the stack
char* pop() {
    if (top == -1) return NULL; // stack is empty
    return stack[top--];
}

// Main function for Task A
void task_a_flow_emergency() {
    printf("\n--- Task A: Flow and Emergency ---\n");

    // Enqueue all 6 flow requests
    for (int i = 0; i < MAX; i++) {
        enqueue(flow_requests[i]);
    }

    // Dequeue and push into stack
    char* item;
    while ((item = dequeue()) != NULL) {
        push(item);
    }

    // Show how emergency stack is handled (LIFO)
    printf("Emergency Adjustment Order (LIFO):\n");
    while ((item = pop()) != NULL) {
        printf("%s\n", item);
    }
}

// ------------------------
// Task B: Power Log (Array)
// ------------------------

char power_log[POWER_LOG_SIZE][SIZE];
int log_index = 0;
int log_count = 0;

// Adds a power output to the log
void log_power(char* output) {
    if (log_count < POWER_LOG_SIZE) {
        strcpy(power_log[log_index++], output);
        log_count++;
    } else {
        // If full, remove oldest and shift left
        printf("Transmitting (Oldest): %s\n", power_log[0]);
        for (int i = 1; i < POWER_LOG_SIZE; i++) {
            strcpy(power_log[i - 1], power_log[i]);
        }
        strcpy(power_log[POWER_LOG_SIZE - 1], output);
    }
}

// Displays current log
void display_power_log() {
    printf("\nCurrent Power Log:\n");
    for (int i = 0; i < log_count; i++) {
        printf("%s\n", power_log[i]);
    }
}

// Main function for Task B
void task_b_power_log() {
    printf("\n--- Task B: Power Output Log ---\n");

    char* outputs[] = {"Pow1", "Pow2", "Pow3", "Pow4", "Pow5", "Pow6", "Pow7"};
    for (int i = 0; i < 7; i++) {
        log_power(outputs[i]);
    }

    display_power_log();
}

// ------------------------
// Task C: Maintenance Tracker
// Singly + Doubly Linked List
// ------------------------

typedef struct SNode {
    char name[SIZE];
    struct SNode* next;
} SNode;

typedef struct DNode {
    char name[SIZE];
    struct DNode* prev;
    struct DNode* next;
} DNode;

SNode* worn_head = NULL;
DNode* repaired_head = NULL;
```

```
// Adds a worn component to the singly linked list
void insert_worn(char* name) {
  SNode* newNode = (SNode*)malloc(sizeof(SNode));
  strcpy(newNode->name, name);
  newNode->next = worn_head;
  worn_head = newNode;
}

// Removes a worn component from the list
void delete_worn(char* name) {
  SNode *temp = worn_head, *prev = NULL;
  while (temp != NULL && strcmp(temp->name, name) != 0) {
    prev = temp;
    temp = temp->next;
  }

  if (temp == NULL) return;

  if (prev == NULL)
    worn_head = temp->next;
  else
    prev->next = temp->next;

  free(temp);
}

// Adds repaired component to doubly linked list
void insert_repaired(char* name) {
  DNode* newNode = (DNode*)malloc(sizeof(DNode));
  strcpy(newNode->name, name);
  newNode->prev = NULL;
  newNode->next = repaired_head;

  if (repaired_head != NULL)
    repaired_head->prev = newNode;

  repaired_head = newNode;
}

// Traverse forward
void traverse_repaired_forward() {
  DNode* temp = repaired_head;
  printf("\nRepaired Components (Forward):\n");
  while (temp != NULL) {
    printf("%s\n", temp->name);
    temp = temp->next;
  }
}
```

```
// Traverse backward
void traverse_repaired_backward() {
    DNode* temp = repaired_head;
    while (temp && temp->next != NULL) {
        temp = temp->next;
    }

    printf("Repaired Components (Backward):\n");
    while (temp != NULL) {
        printf("%s\n", temp->name);
        temp = temp->prev;
    }
}

// Main function for Task C
void task_c_worn_tracker() {
    printf("\n--- Task C: Worn Component Tracker ---\n");

    insert_worn("Turbine");
    insert_worn("Pump");

    delete_worn("Turbine");
    insert_repaired("Turbine");

    traverse_repaired_forward();
    traverse_repaired_backward();
}

// -----------------------
// Task D: Priority Tuning (Circular Linked List)
// -----------------------

typedef struct CNode {
    char name[SIZE];
    struct CNode* next;
} CNode;

CNode* circular_head = NULL;

// Inserts a node into circular linked list
void insert_circular(char* name) {
    CNode* newNode = (CNode*)malloc(sizeof(CNode));
    strcpy(newNode->name, name);

    if (circular_head == NULL) {
        circular_head = newNode;
        newNode->next = circular_head;
    } else {
        CNode* temp = circular_head;
        while (temp->next != circular_head) {
            temp = temp->next;
        }
        temp->next = newNode;
        newNode->next = circular_head;
    }
}

// Traverse twice
void traverse_circular(int rounds) {
    if (!circular_head) return;

    CNode* temp = circular_head;
    printf("\nCircular Priority Tuning:\n");
    for (int i = 0; i < rounds * 2; i++) {
        printf("%s\n", temp->name);
        temp = temp->next;
    }
}

// Main function for Task D
void task_d_priority_tuning() {
    printf("\n--- Task D: Priority Tuning ---\n");

    insert_circular("Gate");
    insert_circular("Spillway");

    traverse_circular(2); // loop twice
}

// ------------------------
// Main Program Starts Here
// ------------------------

int main() {
    task_a_flow_emergency();
    task_b_power_log();
    task_c_worn_tracker();
    task_d_priority_tuning();
    return 0;
}
```