ASSIGNMENT EDSA :- 7

**NAME: - SAGAR**

**ROLL NO: - ME24I1008**

**DATE: - 15/04/2025**

CODE:-

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Task a: Flow and Emergency (Queue and Stack)

#define MAX\_QUEUE 6

#define MAX\_STACK 6

typedef struct {

char\* items[MAX\_QUEUE];

int front, rear;

} Queue;

typedef struct {

char\* items[MAX\_STACK];

int top;

} Stack;

void initQueue(Queue\* q) {

q->front = 0;

q->rear = -1;

}

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

if (q->rear < MAX\_QUEUE - 1) {

q->items[++q->rear] = item;

}

}

char\* dequeue(Queue\* q) {

if (q->front > q->rear) return NULL;

return q->items[q->front++];

}

void initStack(Stack\* s) {

s->top = -1;

}

void push(Stack\* s, char\* item) {

if (s->top < MAX\_STACK - 1) {

s->items[++s->top] = item;

}

}

char\* pop(Stack\* s) {

if (s->top >= 0) {

return s->items[s->top--];

}

return NULL;

}

// Task b: Power Output Log (Array)

#define LOG\_SIZE 5

typedef struct {

char\* outputs[LOG\_SIZE];

int index;

} PowerLog;

void initPowerLog(PowerLog\* log) {

log->index = 0;

for (int i = 0; i < LOG\_SIZE; i++) {

log->outputs[i] = NULL;

}

}

void insertPower(PowerLog\* log, char\* output) {

if (log->index < LOG\_SIZE) {

log->outputs[log->index++] = output;

} else {

printf("Transmitting: %s\n", log->outputs[0]);

for (int i = 0; i < LOG\_SIZE - 1; i++) {

log->outputs[i] = log->outputs[i + 1];

}

log->outputs[LOG\_SIZE - 1] = output;

}

}

void printPowerLog(PowerLog\* log) {

printf("Current Log: [");

for (int i = 0; i < LOG\_SIZE; i++) {

printf("%s", log->outputs[i] ? log->outputs[i] : "null");

if (i < LOG\_SIZE - 1) printf(", ");

}

printf("]\n");

}

// Task c: Worn Component Tracker (Singly and Doubly Linked Lists)

typedef struct SinglyNode {

char\* data;

struct SinglyNode\* next;

} SinglyNode;

typedef struct DoublyNode {

char\* data;

struct DoublyNode\* prev;

struct DoublyNode\* next;

} DoublyNode;

typedef struct {

SinglyNode\* head;

} SinglyLinkedList;

typedef struct {

DoublyNode\* head;

} DoublyLinkedList;

void initSinglyList(SinglyLinkedList\* list) {

list->head = NULL;

}

void insertSingly(SinglyLinkedList\* list, char\* data) {

SinglyNode\* newNode = (SinglyNode\*)malloc(sizeof(SinglyNode));

newNode->data = data;

newNode->next = list->head;

list->head = newNode;

}

void deleteSingly(SinglyLinkedList\* list, char\* data) {

SinglyNode\* current = list->head;

SinglyNode\* prev = NULL;

if (current && strcmp(current->data, data) == 0) {

list->head = current->next;

free(current);

return;

}

while (current && strcmp(current->data, data) != 0) {

prev = current;

current = current->next;

}

if (current) {

prev->next = current->next;

free(current);

}

}

void initDoublyList(DoublyLinkedList\* list) {

list->head = NULL;

}

void insertDoubly(DoublyLinkedList\* list, char\* data) {

DoublyNode\* newNode = (DoublyNode\*)malloc(sizeof(DoublyNode));

newNode->data = data;

newNode->prev = NULL;

newNode->next = list->head;

if (list->head) {

list->head->prev = newNode;

}

list->head = newNode;

}

void traverseForward(DoublyLinkedList\* list) {

DoublyNode\* current = list->head;

printf("Forward: ");

while (current) {

printf("%s ", current->data);

current = current->next;

}

printf("\n");

}

void traverseBackward(DoublyLinkedList\* list) {

DoublyNode\* current = list->head;

if (!current) return;

while (current->next) {

current = current->next;

}

printf("Backward: ");

while (current) {

printf("%s ", current->data);

current = current->prev;

}

printf("\n");

}

// Task d: Priority Tuning (Circular Linked List)

typedef struct CircularNode {

char\* data;

struct CircularNode\* next;

} CircularNode;

typedef struct {

CircularNode\* head;

} CircularLinkedList;

void initCircularList(CircularLinkedList\* list) {

list->head = NULL;

}

void insertCircular(CircularLinkedList\* list, char\* data) {

CircularNode\* newNode = (CircularNode\*)malloc(sizeof(CircularNode));

newNode->data = data;

if (!list->head) {

list->head = newNode;

newNode->next = newNode;

} else {

CircularNode\* current = list->head;

while (current->next != list->head) {

current = current->next;

}

current->next = newNode;

newNode->next = list->head;

}

}

void traverseCircular(CircularLinkedList\* list, int count) {

if (!list->head) return;

CircularNode\* current = list->head;

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

CircularNode\* start = current;

do {

printf("%s ", current->data);

current = current->next;

} while (current != start);

}

printf("\n");

}

int main() {

// Task a: Flow and Emergency

printf("Task a: Flow and Emergency\n");

Queue q;

Stack s;

initQueue(&q);

initStack(&s);

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

for (int i = 0; i < 6; i++) {

enqueue(&q, requests[i]);

}

while (q.front <= q.rear) {

char\* item = dequeue(&q);

push(&s, item);

}

printf("Emergency Adjustment Order:\n");

while (s.top >= 0) {

printf("%s\n", pop(&s));

}

printf("\n");

// Task b: Power Output Log

printf("Task b: Power Output Log\n");

PowerLog log;

initPowerLog(&log);

char\* outputs[] = {"Pow1", "Pow2", "Pow3", "Pow4", "Pow5", "Pow6", "Pow7"};

for (int i = 0; i < 7; i++) {

insertPower(&log, outputs[i]);

printPowerLog(&log);

}

printf("\n");

// Task c: Worn Component Tracker

printf("Task c: Worn Component Tracker\n");

SinglyLinkedList singlyList;

DoublyLinkedList doublyList;

initSinglyList(&singlyList);

initDoublyList(&doublyList);

insertSingly(&singlyList, "Turbine");

insertSingly(&singlyList, "Pump");

deleteSingly(&singlyList, "Turbine");

insertDoubly(&doublyList, "Turbine");

traverseForward(&doublyList);

traverseBackward(&doublyList);

printf("\n");

// Task d: Priority Tuning

printf("Task d: Priority Tuning\n");

CircularLinkedList circularList;

initCircularList(&circularList);

insertCircular(&circularList, "Gate");

insertCircular(&circularList, "Spillway");

traverseCircular(&circularList, 2);

return 0;

}

REPORT OF THIS CODE:-

Task a: Flow and Emergency (Queue and Stack)

Implementation

* **Queue**: Implemented as a circular queue with a fixed size (MAX\_QUEUE=6)
* **Stack**: Implemented as an array-based stack with a fixed size (MAX\_STACK=6)

Strengths

* Clear initialization functions (initQueue, initStack)
* Simple and effective enqueue/dequeue and push/pop operations
* Good demonstration of queue-to-stack transfer logic

Issues

* **Queue overflow**: The enqueue function doesn't handle cases where the queue is full (rear reaches MAX\_QUEUE-1)
* **Stack overflow**: Similarly, push doesn't handle stack overflow
* **No bounds checking**: Both data structures could benefit from isFull/isEmpty checks

Recommendation

* Add boundary condition checks
* Consider dynamic resizing for more practical use

Task b: Power Output Log (Array)

Implementation

* Fixed-size circular buffer (LOG\_SIZE=5) for power output logging
* Implements automatic transmission of oldest entry when full

Strengths

* Effective use of array as circular buffer
* Clear printing function
* Automatic handling of overflow by transmitting oldest data

Issues

* Memory management: The code stores pointers but doesn't handle string allocation/deallocation
* No protection against NULL pointer dereferencing in insertPower

Recommendation

* Add string duplication for safer memory management
* Consider adding error handling for NULL inputs

Task c: Worn Component Tracker (Singly and Doubly Linked Lists)

Implementation

* Both singly and doubly linked lists for component tracking
* Basic insertion and deletion operations

Strengths

* Clear demonstration of both list types
* Proper memory deallocation in deleteSingly
* Correct implementation of forward/backward traversal

Issues

* No memory allocation check in insertion functions
* deleteSingly could segfault if item not found (prev would be NULL)
* No function to free entire lists

Recommendation

* Add error checking for malloc
* Improve deletion safety
* Add list destruction functions

Task d: Priority Tuning (Circular Linked List)

Implementation

* Circular linked list for priority management
* Insertion and traversal functions

Strengths

* Correct circular list implementation
* Flexible traversal with count parameter

Issues

* No deletion functionality
* Memory leak potential (no cleanup function)
* Traversal could be infinite if count is very large

Recommendation

* Add deletion capability
* Implement list cleanup
* Consider adding traversal limits

General Observations

1. **Memory Management**: The code frequently stores string pointers without managing the underlying memory. This could lead to issues if the original strings are modified or freed.
2. **Error Handling**: Most functions lack error handling for edge cases (NULL inputs, allocation failures, etc.)
3. **Testing**: The main function provides good test cases but could benefit from more edge case testing.
4. **Modularity**: The code is well-organized into distinct sections for each task.
5. **Documentation**: While the structure is clear, adding comments explaining the purpose of each function would improve maintainability.

Overall Assessment

The code demonstrates a solid understanding of fundamental data structures and their applications. With added error handling and memory management improvements, it would be production-ready. The separation of concerns into different tasks makes the code easy to understand and maintain.

**Rating**: 8/10 (Excellent demonstration of concepts, could benefit from more robust error handling and memory management)

OUTPUT OF THIS CODE:-

Task a: Flow and Emergency

Emergency Adjustment Order:

Valve

Pump

Reservoir

Spillway

Gate

Turbine

Task b: Power Output Log

Current Log: [Pow1, null, null, null, null]

Current Log: [Pow1, Pow2, null, null, null]

Current Log: [Pow1, Pow2, Pow3, null, null]

Current Log: [Pow1, Pow2, Pow3, Pow4, null]

Current Log: [Pow1, Pow2, Pow3, Pow4, Pow5]

Transmitting: Pow1

Current Log: [Pow2, Pow3, Pow4, Pow5, Pow6]

Transmitting: Pow2

Current Log: [Pow3, Pow4, Pow5, Pow6, Pow7]

Task c: Worn Component Tracker

Forward: Turbine

Backward: Turbine

Task d: Priority Tuning

Gate Spillway Gate Spillway

=== Code Execution Successful ===

Conclusion

This C program effectively demonstrates the practical application of fundamental data structures—**queues, stacks, arrays, singly/doubly linked lists, and circular linked lists**—to simulate key operations in a **hydroelectric dam management system**.

**Key Strengths**

✔ **Well-Structured & Modular**: The code is logically divided into four distinct tasks, each showcasing a different data structure.  
✔ **Correct Implementations**: The core functionalities (enqueue/dequeue, push/pop, circular buffer management, linked list traversals) work as expected.  
✔ **Good Demonstration of Concepts**: The program effectively illustrates how different data structures can be applied in real-world scenarios (emergency adjustments, power logging, component tracking, priority tuning).

**Areas for Improvement**

⚠ **Memory Management**: The code stores raw string pointers without duplication or cleanup, risking memory leaks or undefined behavior.  
⚠ **Error Handling**: Missing checks for NULL inputs, allocation failures, and boundary conditions (e.g., full queue/stack).  
⚠ **Missing Features**: Some functionalities (like list cleanup, circular list deletion) are absent but would be necessary for a complete system.

**Final Verdict**

This program serves as an **excellent educational example** of data structure implementations in C. With **enhanced memory safety, error handling, and additional features**, it could evolve into a robust simulation for real-world dam management systems.

**Recommendation**:

* **Add**malloc**checks & string duplication (**strdup**)** for safer memory handling.
* **Implement boundary checks** (e.g., isFull, isEmpty) for queues and stacks.
* **Include cleanup functions** to prevent memory leaks.