Elementary data structure and logical thinking

Roll No:-ME24B1016
Name:-Suranjana Mary
assignment-Qn-4

```c
#include <stdio.h>//header lines like Standard Input Output Header
#include <stdlib.h>//Standard Library Header
#include <string.h>//header to provide string-related functions.
#define MAX 10//global declarations
#define storage_arr 5
#define name_len 20
//parts arrive via rover and queue for construction.
char queue[MAX][name_len];//defining queue
int front = -1, rear = -1;

void enqueue(char part[])//function to add parts into the queue
{
    if (rear == MAX - 1)
    {
        printf("Queue is full!\n");
        return;
    }
    if (front == -1) front = 0;
    rear++;
    strcpy(queue[rear], part);
}

char* dequeue()//function to remove parts from the queue
{
    if (front == -1 || front > rear) return NULL;
    return queue[front++];
}

//defining stack for bot task manager.
char stack[MAX][name_len];
int top = -1;

void push(char part[]) // to push elements into the stack
{
    if (top == MAX - 1)
    {
        printf("Stack is full!\n");
        return;
    }
    top++;
    strcpy(stack[top], part);
}
```

```
char* pop() //to remove elements from the stack
{
    if (top == -1) return NULL;
    return stack[top--];
}

// defining array in order to work as the assembly storage unit
char storage[storage_arr][name_len];
int insert = 0;

void insertHabitat(char hab[])
{
    strcpy(storage[insert], hab);
    insert = (insert + 1) % storage_arr; // if full,overwriting the oldest
habitat place
}

//defining singly linked list for damaged habitat
typedef struct SNode {
    char name[name_len];
    struct SNode* next;
} SNode;

//defining doubly linked list for repaired habitats
typedef struct DNode {
    char name[name_len];
    struct DNode* next;
    struct DNode* prev;
} DNode;

//defining circular linked list for priority upgrades
typedef struct CNode {
    char name[name_len];
    struct CNode* next;
} CNode;
//linked lists as damaged habitat tracker
SNode* damaged_head = NULL;
DNode* repaired_head = NULL;
DNode* repaired_tail = NULL;
CNode* circular_head = NULL;
//function to insert damaged habitat using singly linked list
void insertDamaged(char hab[])
{
```

```
    SNode* new_node = (SNode*)malloc(sizeof(SNode));
    strcpy(new_node->name, hab);
    new_node->next = damaged_head;
    damaged_head = new_node;
}
//function to move the repaired habitats to doubly linkes list for
inspection
void moverepaired(char hab[])
{
    SNode *prev = NULL, *curr = damaged_head;
    while (curr && strcmp(curr->name, hab) != 0) {
        prev = curr;
        curr = curr->next;
    }
    if (!curr) {
        printf("Habitat %s not found in damaged list.\n", hab);
        return;
    }
    if (prev) prev->next = curr->next;
    else damaged_head = curr->next;

    DNode* new_node = (DNode*)malloc(sizeof(DNode));
    strcpy(new_node->name, hab);
    new_node->next = NULL;
    new_node->prev = repaired_tail;

    if (!repaired_head) {
        repaired_head = repaired_tail = new_node;
    } else {
        repaired_tail->next = new_node;
        repaired_tail = new_node;
    }
    free(curr);
}
//function to traverse forward and backward
void traverserepaired()
{
    DNode* curr = repaired_head;
    printf("\nFORWARD: Repaired Habitats:\n");
    while (curr)
    {
        printf("%s\n", curr->name);
        curr = curr->next;
```

```
    }
    printf("BACKWARD:\n");
    curr = repaired_tail;
    while (curr) {
        printf("%s\n", curr->name);
        curr = curr->prev;
    }
}
//function to insert the habitats into circular linked list fro priority
upgrades
void insertcircular(char hab[])
{
    CNode* new_node = (CNode*)malloc(sizeof(CNode));
    strcpy(new_node->name, hab);
    if (!circular_head)
    {
        circular_head = new_node;
        new_node->next = new_node;
    }
    else
    {
        CNode* temp = circular_head;
        while (temp->next != circular_head)
            temp = temp->next;
        temp->next = new_node;
        new_node->next = circular_head;
    }
}
//fucntio to traverse twice in circular linkd list
void traverseCircular(int times)
{
    printf("\nPRIORITY UPGRADE CYCLE:\n");
    if (!circular_head) return;
    CNode* curr = circular_head;
    for (int i = 0; i < times; i++)
    {
        printf("%s\n", curr->name);
        curr = curr->next;
    }
}


int main() {
```

```
    int i;
    char part[name_len];

    printf("Enter 6 Parts for Queue (e.g., Wall,
Roof,Door,Window,Airlock,Vent)\n");
    for (i = 0; i < 6; i++)
    {
        scanf("%s", part);
        enqueue(part);
    }

    // Transfer from queue to stack
    while (front <= rear)
    {
        char* temp = dequeue();
        if (temp) push(temp);
    }

    printf("\nCONSTRUCTION ORDER (LIFO):\n");
    while (top >= 0)
    {
        printf("%s\n", pop());
    }


    printf("\nEnter 7 Habitat Names (eg.hab1,hab2,hab3,.....,hab7\n");
    for (i = 0; i < 7; i++)
    {
        scanf("%s", part);
        insertHabitat(part);
    }
    printf("\nFINAL ASSEMBLY STORAGE:\n");
    for (i = 0; i < storage_arr; i++)
    {
        printf("%s\n", storage[i]);
    }
    printf("\nEnter 2 Damaged Habitats\n");
    for (i = 0; i < 2; i++)
    {
        scanf("%s", part);
        insertDamaged(part);
    }
```

```
    printf("\nEnter the name of the habitat to be repaired: ");
    scanf("%s", part);
    moverepaired(part);
    traverserepaired();
    printf("\nEnter 2 Priority Upgrade Habitats\n");
    for (i = 0; i < 2; i++)
    {
        scanf("%s", part);
        insertcircular(part);
    }
    traverseCircular(4);//to show the circular nature here 4 has been used
    return 0;
}
//"Bonus questions and answers"
//1.Why LIFO fits?
//Ans:LIFO that is last in first out is used here because the last
component added (like Vents or Airlocks) needs to be installed first to
seal the structure tightly. This ensures that critical parts responsible
for pressure and atmosphere containment are placed immediately before the
habitat is closed.thats why we use stacks here.
//2.Suggest a reason for occupation?
//Ans:On Mars, space is limited and every module counts. As new settlers
arrive or newer, upgraded habitats are built, older modules are overwritten
to make room for the new ones. This ensures the colony always has the
latest tech and comfort for its growing population.
//3.Propose damage and fix.
//Ans:Assuming that habitat 3's door was cracked during a Martian dust
storm, compromising air-tightness. Repair bots quickly sealed the damage
and moved the habitat into a repaired queue for inspection, ensuring it met
colony safety standards before re-entry.
//4.Invent an upgrade.
//Ans:like Hab1 is being fitted with solar panels for better energy
independence, Hab4 receives a greenhouse dome, allowing settlers to grow
crops indoors,hab 6 is fitted with a recreational centre so people there
stay fit and happy.
```

OUTPUT

```
Enter 6 Parts for Queue (e.g., Wall, Roof,Door,Window,Airlock,Vent)
wall
roof
door
window
airlock
vent

CONSTRUCTION ORDER (LIFO):
vent
airlock
window
door
roof
wall

Enter 7 Habitat Names (eg.hab1,hab2,hab3,.....,hab7
hab1
hab2
hab3
hab4
hab6
hab7

FINAL ASSEMBLY STORAGE:
hab6
hab7
hab3
hab4
hab5

Enter 2 Damaged Habitats
hab3
hab4

Enter the name of the habitat to be repaired: hab3

FORWARD: Repaired Habitats:
hab3
BACKWARD:
hab3
```

```
Enter 2 Priority Upgrade Habitats
hab2
hab1

PRIORITY UPGRADE CYCLE:
hab2
hab1
hab2
hab1
```

REPORT

Mars Habitat Builder Bot

Problem Statement

As space exploration advances, building sustainable living modules on Mars is a critical challenge. The project aims to simulate a Mars Habitat Builder Bot using C programming and different data structures to manage the assembly, repair, and upgrade of habitats efficiently.

Objectives

1. **Simulate a part-handling and assembly system** where Mars habitat modules arrive via rover and get constructed using queues and stacks.

2. **Store and manage assembled habitats** using an efficient storage system using arrays.

3. **Track damaged habitats** and repair them using linked lists.

4. **Manage upgraded habitats** using a circular priority system.

Why I Chose These Data Structures

Each data structure in this program is carefully selected based on its behavior and suitability for the task:

- **Queue (FIFO)** – Represents the real-world scenario where parts arrive via rover and are handled in the order of arrival.

- **Stack (LIFO)** – Used for the construction process, where the last item added is the first to be used (like sealing components).

- **Array** – Acts as storage for habitats with limited space, overwriting the oldest entries when full using a circular array.

- **Singly Linked List** – Tracks damaged habitats. Easy to add and search through.
- **Doubly Linked List** – Manages repaired habitats for inspection, allowing both forward and backward traversal.
- **Circular Linked List** – Perfect for priority upgrades, where we cycle through the list continuously.

Working of the code

1. User Inputs Part Names (e.g., Wall, Roof, Door, etc.) added to a queue.
2. Parts are then transferred to a stack to simulate LIFO construction order.
3. Stack elements are popped and displayed in reverse construction order.
4. The user enters habitat names, which are stored in a fixed-size array using circular.
5. The user then enters damaged habitats, stored in a singly linked list.
6. A damaged habitat is repaired and moved to a doubly linked list, allowing inspection forward and backward.
7. The user enters habitats for priority upgrades, which are added to a circular linked list, and it is traversed cyclically to show continuous upgrading.

Variables and Functions Used

Global Constants

- `MAX` → Max size for queue/stack = 10.
- `storage_arr` → Max storage size = 5.
- `name_len` → Max length of strings (habitat or part name) = 20.

Queue

- `queue[MAX][name_len]`, `front`, `rear`
- `enqueue(char part[])`: Adds part to the queue.
- `dequeue()`: Removes the oldest part from the queue.

Stack

- `stack[MAX][name_len]`, `top`
- `push(char part[])`: Adds part to the stack.
- `pop()`: Removes the most recently added part.

Array (Storage)

- `storage[storage_arr][name_len]`, `insert`
- `insertHabitat(char hab[])`: Stores habitats in circular manner.

Singly Linked List (Damaged)

- `SNode` struct with `name` and `next`
- `insertDamaged(char hab[])`: Adds damaged habitats.

Doubly Linked List (Repaired)

- `DNode` struct with `name`, `next`, and `prev`
- `moverepaired(char hab[])`: Moves habitat from damaged to repaired.
- `traverserepaired()`: Traverses forward and backward.

Circular Linked List (Upgrades)

- `CNode` struct with `name` and `next`
- `insertcircular(char hab[])`: Adds habitat for upgrade.
- `traverseCircular(int times)`: Traverses the circular list n times.

Time Complexity

- **Queue/Stack operations** → O(1) per operation.
- **Array insertions** → O(1)
- **Singly Linked List search/delete** → O(n)
- **Doubly Linked List traversal** → O(n)
- **Circular Linked List traversal** → O(n), controlled by the number of cycles.

Overall, the simulation runs efficiently within **O(n)** time for each module.

Conclusion

This project successfully demonstrates how **real-world logistics and systems on Mars** could be modeled using basic **C programming and data structures**. It highlights not just code efficiency but how structural decisions can solve practical problems like construction order, damage tracking, storage limitation, and upgrade cycles.