# Assignment Title:- Deep Sea Rover Repair Station

Name: Srinivasan

Roll No: ME24I1015

Date: 15-04-2025

## Code Implementation

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Structure to hold a rover part

typedef struct {

char partName[20];

} Part;

// Structure to hold a rover

typedef struct {

char roverName[20];

int isFaulty;

int needsUpgrade;

} Rover;

// Queue Node for Part Delivery

typedef struct QueueNode {

Part data;

struct QueueNode\* next;

} QueueNode;

typedef struct {

QueueNode\* front;

QueueNode\* rear;

} Queue;

// Array for Assembly Storage (max 7 rovers)

#define MAX\_STORAGE 7

Rover roverArray[MAX\_STORAGE];

int roverCount = 0;

// Singly Linked List for Faulty Rovers

typedef struct SinglyListNode {

Rover data;

struct SinglyListNode\* next;

} SinglyListNode;

// Doubly Linked List for Inspected Rovers

typedef struct DoublyListNode {

Rover data;

struct DoublyListNode\* prev;

struct DoublyListNode\* next;

} DoublyListNode;

// Circular Linked List for Priority Upgrades

typedef struct CircularNode {

Rover data;

struct CircularNode\* next;

} CircularNode;

// Queue Functions

void initializeQueue(Queue\* q) {

q->front = q->rear = NULL;

}

void addToQueue(Queue\* q, Part p) {

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

strcpy(newNode->data.partName, p.partName);

newNode->next = NULL;

if (q->rear == NULL) {

q->front = q->rear = newNode;

} else {

q->rear->next = newNode;

q->rear = newNode;

}

}

Part removeFromQueue(Queue\* q) {

if (q->front == NULL) {

printf("No parts left in queue!\n");

Part empty = {"None"};

return empty;

}

QueueNode\* temp = q->front;

Part p = temp->data;

q->front = q->front->next;

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

free(temp);

return p;

}

void pushToStack(Part p) {

printf("Stacking part for arm: %s\n", p.partName); // Simulating LIFO stack

}

// Assembly Storage Functions

void storeRover(Rover r) {

if (roverCount < MAX\_STORAGE) {

roverArray[roverCount++] = r;

} else {

printf("Storage full, removing oldest rover: %s\n", roverArray[0].roverName);

for (int i = 0; i < MAX\_STORAGE - 1; i++) {

roverArray[i] = roverArray[i + 1];

}

roverArray[--roverCount] = r;

}

}

void handleOverflow() {

printf("Overflow for Rov6 and Rov7 - deploying due to saltwater corrosion risk.\n");

}

// Faulty Rover Tracker Functions

SinglyListNode\* faultyHead = NULL;

void addFaultyRover(Rover r) {

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

newNode->data = r;

newNode->next = faultyHead;

faultyHead = newNode;

}

void moveToInspected(DoublyListNode\*\* inspectedHead, Rover r) {

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

newNode->data = r;

newNode->prev = NULL;

newNode->next = \*inspectedHead;

if (\*inspectedHead != NULL) (\*inspectedHead)->prev = newNode;

\*inspectedHead = newNode;

}

// Priority Upgrade Functions

CircularNode\* upgradeHead = NULL;

void addUpgrade(Rover r) {

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

newNode->data = r;

if (upgradeHead == NULL) {

newNode->next = newNode; // Circular link to self

upgradeHead = newNode;

} else {

CircularNode\* last = upgradeHead;

while (last->next != upgradeHead) last = last->next;

last->next = newNode;

newNode->next = upgradeHead;

}

}

void processUpgrades() {

if (upgradeHead == NULL) return;

CircularNode\* current = upgradeHead;

do {

printf("Processing upgrade for: %s\n", current->data.roverName);

current = current->next;

} while (current != upgradeHead);

}

// Main function to run the simulation

int main() {

Queue partQueue;

initializeQueue(&partQueue);

DoublyListNode\* inspectedHead = NULL;

// a) Part Delivery and Arm

Part parts[] = {{"Sonar"}, {"Drill"}, {"Wheel"}, {"Camera"}, {"Arm"}, {"Battery"}};

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

addToQueue(&partQueue, parts[i]);

}

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

Part p = removeFromQueue(&partQueue);

pushToStack(p);

}

// b) Assembly Storage Unit

Rover rovers[] = {{"Rov1"}, {"Rov2"}, {"Rov3"}, {"Rov4"}, {"Rov5"}, {"Rov6"}, {"Rov7"}};

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

storeRover(rovers[i]);

}

handleOverflow();

// c) Faulty Rover Tracker

rovers[2].isFaulty = 1; // Rov3 is faulty

rovers[5].isFaulty = 1; // Rov6 is faulty

addFaultyRover(rovers[2]);

addFaultyRover(rovers[5]);

moveToInspected(&inspectedHead, rovers[2]);

// d) Priority Upgrades

rovers[0].needsUpgrade = 1; // Rov1 needs upgrade

rovers[3].needsUpgrade = 1; // Rov4 needs upgrade

addUpgrade(rovers[0]);

addUpgrade(rovers[3]);

processUpgrades();

// Basic memory cleanup

while (faultyHead != NULL) {

SinglyListNode\* temp = faultyHead;

faultyHead = faultyHead->next;

free(temp);

}

while (inspectedHead != NULL) {

DoublyListNode\* temp = inspectedHead;

inspectedHead = inspectedHead->next;

free(temp);

}

if (upgradeHead != NULL) {

CircularNode\* last = upgradeHead;

while (last->next != upgradeHead) last = last->next;

last->next = NULL;

while (upgradeHead != NULL) {

CircularNode\* temp = upgradeHead;

upgradeHead = upgradeHead->next;

free(temp);

}

}

return 0;

}

Problem solving:-

I am solving the challenge of designing a robust repair station for deep-sea rovers, where Parts must be processed efficiently despite limited resources.Storage is constrained, requiring overflow management in a corrosive environment.Faulty rovers need tracking and repair tracking with flexibility.Upgrades must be prioritized and repeatedly checked in a continuous cycle.The code integrates these components into a cohesive simulation, addressing real-world deep-sea maintenance issues like corrosion, navigation readiness, and operational upgrades, as outlined in the assignment objectives.

Key Objective:-

The key objective of the Deep-Sea Rover Repair Station assignment is to simulate the repair process, tracking part flow, storage, and upgrades for maintaining rover parts in a deep-sea environment. Managing the storage and deployment of repaired rovers in an array-based system.Tracking faulty rovers and moving them to a repaired list using linked lists. This objective ensures efficient repair, storage, and upgrade management to support deep-sea rover operations.

Choosing this specific data structure:-

Each task (a, b, c, d) specifies a data structure (queue, stack, array, linked lists), guiding your choices to meet the simulation objectives.Deep-Sea Context: The structures address real-world constraints like limited storage (array), corrosion risk (array overflow), fault tracking (linked lists), and priority management (circular list).Efficiency and Simplicity: You selected structures that balance functionality with ease of implementation, such as using an array for fixed storage and linked lists for dynamic tracking, avoiding overly complex alternatives (e.g., balanced trees).

Overall Efficiency GainsTask-Specific Optimization:-

Each data structure was tailored to its task (FIFO for delivery, LIFO for arm, fixed storage for assembly, dynamic tracking for faults, bidirectional for inspection, and cyclic for upgrades), reducing the need for generic or overcomplicated solutions.Time Complexity: Operations like enqueue/dequeue (O(1)), array access (O(1)), linked list insertion (O(1)), and circular traversal (O(n) per cycle) were efficient for the small-scale simulation (6 parts, 7 rovers).Space Efficiency: Dynamic structures (linked lists) avoided over-allocation, while the fixed array minimized memory use, critical for a resource-constrained deep-sea environment.

Step-by-Step Execution:-

1)Initialization: partQueue initialized (empty), inspectedHead set to NULL.

2)Task a (Part Delivery): Enqueue 6 parts ("Sonar" to "Battery"); dequeue and stack (LIFO) with output: "Stacking part for arm: Sonar" to "Drill".

3)Task b (Storage): Store 7 rovers ("Rov1" to "Rov7"), remove "Rov1" and "Rov2" on overflow, output: "Storage full, removing oldest rover: Rov1", "Rov2", "Overflow for Rov6 and Rov7..."

4)Task c (Faulty Tracker): Mark "Rov3" and "Rov7" as faulty, add to faultyHead, move "Rov3" to inspectedHead (no output).

5)Task d (Upgrades): Mark "Rov3" and "Rov6" for upgrade, add to upgradeHead, process twice with output: "Processing upgrade for: Rov3", "Rov6" (x2).

6)Cleanup: Free memory for faultyHead, inspectedHead, upgradeHead (no output).

Functions and their purposes

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Function Name | Parameters | Return Type | Purpose | Task |
| initializeQueue | Queue\* q | Void | Initializes the queue by setting both front and rear pointers to NULL, preparing it to receive parts via a submersible. | a |
| addToQueue | Queue\* q,Part p | Void | Adds a new part (e.g., "Sonar") to the rear of the queue, simulating the arrival of parts in FIFO order from the submersible. | a |
| removeFromQueue | Queue\* q | Part | Removes and returns the part at the front of the queue, enabling FIFO processing by the robotic arm, with memory cleanup. | a |
| pushToStack | Part p | Void | Simulates the robotic arm stacking a part in LIFO order by printing its name, fulfilling the LIFO processing requirement. | a |
| storeRover | Rover r | Void | Stores a rover in the roverArray, shifting out the oldest rover if full, to manage the 5-slot storage with overflow. | b |
| handleOverflow | None | Void | Notifies the system of overflow for "Rov6" and "Rov7", indicating deployment due to saltwater corrosion risk. | b |
| addFaultyRover | Rover r | Void | Adds a faulty rover (e.g., "Rov3") to the faultyHead singly linked list, tracking faults dynamically. | c |
| moveToInspected | DoublyListNode\*\* inspectedHead, Rover r | Void | Moves a repaired rover (e.g., "Rov3") to the inspectedHead doubly linked list, enabling bidirectional tracking. | c |
| addUpgrade | Rover r | Void | Adds a rover needing an upgrade (e.g., "Rov1") to the upgradeHead circular linked list, supporting priority management | d |
| processUpgrades | None | Void | Traverses the circular linked list twice, printing each rover name to simulate upgrade processing for priority rovers. | d |
| main | None | int | Orchestrates the entire simulation by initializing data structures, executing tasks a, b, c, d, and cleaning up memory. | a,b,c,d |

Variables and Their Purposes:-

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Variable Name | Type | Scope | Purpose | Task |
| Global Variables |  |  |  |  |
| roverArray[MAX\_STORAGE] | Array of Rover structures (size 7) | Global | Stores up to 5 repaired rovers, with overflow handling for the Assembly Storage Unit. | b |
| roverCount | Integer (initially 0) | Global | Tracks the number of rovers currently stored in roverArray to manage capacity.b | b |
| faultyHead | Pointer to SinglyListNode (initially NULL) | Global | Serves as the head of the singly linked list to track rovers identified as faulty. | c |
| upgradeHead | Pointer to CircularNode (initially NULL) | Global | Acts as the head of the circular linked list to manage rovers needing priority upgrades. | d |
| Structures |  |  |  |  |
| Part | Structure with partName[20] (char array) | Define d | Holds the name of a rover part (e.g., "Sonar") for delivery and processing. | a |
| Rover | Structure with roverName[20], isFaulty, needsUpgrade (char array, ints) | Define d | Contains rover details including name, fault status, and upgrade need for various tasks. | b, c, d |
| QueueNode | Structure with data (Part) and next (pointer to QueueNode) | Define d | Represents a node in the queue, holding a part and linking to the next node. | a |
| Queue | Structure with front and rear (pointers to QueueNode) | Define d | Manages the queue structure for part delivery with front and rear pointers. | a |
| SinglyListNode | Structure with data (Rover) and next (pointer to SinglyListNode) | Define d | Defines a node for the singly linked list to track faulty rovers. | c |
| DoublyListNode | Structure with data (Rover), prev, next (pointers to DoublyListNode) | Define d | Defines a node for the doubly linked list to track inspected rovers with bidirectional links. | c |
| CircularNode | **Structure with data (Rover) and next (pointer to CircularNode)** | **Define d** | Defines a node for the circular linked list to manage priority upgrades with circular linking. | d |

For local variables,

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Local Variables |  |  |  |  |
| partQueue | Queue variable | Local (main) | Instance of the queue used to simulate part delivery from the submersible.a | a |
| inspectedHead | DoublyListNode\* pointer (initially NULL) | Local (main) | Pointer to the head of the doubly linked list for inspected rovers. | c |
| parts[] | Array of 6 Part structures ({"Sonar", "Drill", "Wheel", "Camera", "Arm", "Battery"}) | Local (main) | Holds the 6 parts to be enqueued and processed by the arm. | a |
| rovers[] | Array of 7 Rover structures ({"Rov1", "Rov2", "Rov3", "Rov4", "Rov5", "Rov6", "Rov7"}) | Local (main) | Holds the 7 rovers for storage, fault tracking, and upgrade management.b, c, d | b, c, d |
| i | Integer loop variable | Local (main) | Used as a counter to iterate over parts and rovers arrays in loops. | a, b, c, d |
| p | Part variable | Local (main) | Temporary variable to hold the part returned by removeFromQueue for stacking. | a |