# EDSA Assignment Q8: Earthquake Rescue Robot Coordinator

Name: Ayush Gupta  
Roll Number: ME24B1076  
Date: 15/04/2025

**PROBLEM STATEMENT:**

Designing an Earthquake Rescue Robot Coordinator to deploy robots (e.g., "Scanner" , "Digger" , "Lift" , "Light" , "Drone") for disaster relief.

**KEY OBJECTIVES:**

* Implement efficient mission queuing and urgency management.
* Maintain a fixed-size mission log with overwrite feature.
* Track damaged vs repaired robots using appropriate data structures.
* Redeploy high-priority robots in a cyclic pattern.

**DESIGN EXPLANATION:**

**Why These Data Structures ?**

* **Circular Queue**: Used for mission allocation to maintain FIFO order in limited space.
* **Stack**: Manages urgent missions in LIFO order, ensuring last-in tasks are executed first.
* **Fixed-Size Array**: Rescue log stores limited entries, automatically replacing old logs.
* **Singly Linked List**: Efficiently adds/removes damaged robots.
* **Doubly Linked List**: Allows forward/backward traversal of repaired robots for inspection.
* **Circular Linked List**: Allows cyclic traversal for redeployment.

**How they are efficient ?**

* **Queue/Stack**: O(1) for insert/remove.
* **Linked Lists**: Dynamic, no reallocation needed.
* **Circular List**: Natural for repeated rounds without reset logic.

**LOGIC OF THE CODE:**

**Task 1: Mission and Urgency**

* Robots are enqueued into a **circular queue**.
* Urgent task’s robots are dequeued and pushed into a **stack**.
* Stack is then popped to execute urgent tasks.

**Task 2: Rescue Log**

* New missions are logged in a fixed array.
* On overflow, the oldest log is overwritten by shifting all entries left.

**Task 3: Damaged and Repaired Tracking**

* Damaged robots added to a **singly linked list**.
* Once repaired, removed from damaged list and added to a **doubly linked list**.
* Traversals are done in both directions.

**Task 4: Priority Redeployment**

* Robots needing urgent action are added to a **circular linked list**.
* Traversed multiple times to simulate periodic redeployment.

**KEY VARIABLES AND FUNCTIONS:**

***Task 1: Mission and Urgency*  
Variables:**

* **Cqueue: Circular queue to manage incoming missions.**
* **Stack: Stack to prioritize urgent missions.**

**Functions:**

* **init\_queue(), init\_stack(): Initialize queue and stack pointers.**
* **enqueue(): Add a robot to the mission queue.**
* **dequeue(): Remove a robot from the mission queue.**
* **push(): Push a robot into the urgent stack.**
* **pop(): Pop a robot out of the urgent stack.**

***Task 2: Rescue Log*Variables:**

* **Log: Stores the rescue mission history in a fixed-size array**

**Functions**

* **init\_log(): Initializes the log counter.**
* **log\_mission(): Adds a new mission to the log and if full, then removes the oldest one by shifting the array.**

***Task 3: Damaged Robot Tracker*  
Variables:**

* **singly\_node: Node of singly linked list for damaged robots.**
* **doubly\_node: Node of doubly linked list for repaired robots.**

**Functions:**

* **createsingly(): Allocates and initializes a new damaged robot node.**
* **insert\_damaged(): Adds a damaged robot to the end of the singly linked list.**
* **remove\_damaged(): Removes a damaged robot from the list when repaired.**
* **create\_doubly(): Allocates and initializes a new repaired robot node.**
* **insert\_repaired(): Adds a robot to the doubly linked list.**
* **traverse\_forward(): Traverses repaired robot list forward.**
* **traverse\_backward(): Traverses repaired robot list backward.**

***Task 4: Priority Redeployment*  
Variables:**

* **circular\_node: Node of circular linked list for priority redeployment.**

**Functions:**

* **create\_circular(): Creates a new circular list node.**
* **insert\_priority(): Adds a robot to the circular list.**
* **traverse\_circular(): Cycles through the priority redeployment list for a given number of rounds.**

**ANSWERS TO CREATIVITY QUESTIONS**

**Q1. Why LIFO fits for urgent tasks deployment ?**

LIFO fits for urgent tasks deployment because the most recent urgent task will need immediate attention to be in action. e.g. The cutter robot added at last must be deployed first to clear debris before other robots can access affected areas.

**Q2. Reason for reporting:**

Reporting completed missions will allow authorities to update statistics & manage manpower and resources to active missions rather than completed ones.

**Q3. Damage and Fix:**

Suppose digger robot's hydraulic pipe got severely damaged byfalling debris which requires  
complete of its pressure lines and valves. Engineers need to reinforced the frame with carbon fiber to improve durability.   
 **Q4. Redeployment Tweak:**

The Lift robot got thermal imaging camera which is capable of detecting heat through concrete up to 2.5 meters thick, enabling it to prioritize areas with confirmed signs of alive trapped survivors.

***CODE:***

// Name: Ayush Gupta

// Roll Number: ME24B1076

#include <stdio.h>

#include <string.h>

#include <stdlib.h>

#define max 6

#define name 8

// Task 1: Mission and Urgency

typedef struct {

    char robo[max][name];

    int fr;

    int rr;

} Cqueue;  // Circular queue for missions

typedef struct {

    char robo[max][name];

    int top;

} Stack;  // Stack for urgent missions

// For circular queue

void init\_queue(Cqueue \*q)

{

    q->fr = -1;

    q->rr = -1;

}

int queue\_full(Cqueue \*q)

{

    return (q->rr + 1) % max == q->fr;

}

int queue\_empty(Cqueue \*q)

{

    return q->fr == -1;

}

void enqueue(Cqueue \*q, char \*robo)

{

    if (queue\_full(q))

    {

        printf("Queue overflow!\n");

        return;

    }

    else if (queue\_empty(q))

    {

        // Intialize both to 0

        q->fr = 0;

        q->rr = 0;

    }

    else

    {

        q->rr = (q->rr + 1) % max;

    }

    strcpy(q->robo[q->rr], robo);  // Copying strings

    printf("%s enqueued.\n", robo);

}

char\* dequeue(Cqueue \*q)

{

    if (queue\_empty(q))

    {

        printf("Queue underflow!\n");

        return "";

    }

    char \*robo = q->robo[q->fr];

    if (q->fr == q->rr) // Its the last element

    {

        q->fr = -1;

        q->rr = -1;

    }

    else

    {

        q->fr = (q->fr + 1) % max;

    }

    return robo;

}

// For stack

void init\_stack(Stack \*s)

{

    s->top = -1;

}

int stack\_empty(Stack \*s)

{

    return s->top == -1;

}

int stack\_full(Stack \*s)

{

    return s->top == max - 1;

}

void push(Stack \*s, char \*robo)

{

    if (stack\_full(s))

    {

        printf("Stack overflow!\n");

        return;

    }

    s->top++;

    strcpy(s->robo[s->top], robo);

    printf("%s pushed.\n", robo);

}

char\* pop(Stack \*s)

{

    if (stack\_empty(s))

    {

        printf("Stack underflow!\n");

        return "";

    }

    char \*robo = s->robo[s->top];

    s->top--;

    return robo;

}

// Executing task 1

void mission\_and\_urgency()

{

    Cqueue q;

    Stack s;

    init\_queue(&q);

    init\_stack(&s);

    enqueue(&q, "Scanner");

    enqueue(&q, "Digger");

    enqueue(&q, "Lift");

    enqueue(&q, "Light");

    enqueue(&q, "Drone");

    enqueue(&q, "Cutter");

    printf("\n");

    while (!queue\_empty(&q))

    {

        // Dequeue and push urgent tasks

        push(&s, dequeue(&q));

    }

    printf("\n");

    while (!stack\_empty(&s))

    {

        printf("Executing urgent tasks: %s\n", pop(&s));

    }

    printf("\n");

}

// Task 2: Rescue log unit

typedef struct {

    char missions[max][name];

    int counter;

} Log;

void init\_log(Log \*log)

{

    log->counter = 0;

}

void log\_mission(Log \*log, char \*mission)

{

    if (log->counter < 6)

    {

        strcpy(log->missions[log->counter], mission);

        printf("%s logged (%d)\n", mission, log->counter);

        log->counter++;

    }

    else  // If array is full

    {

        printf("\nReporting oldest mission: %s\n", log->missions[0]);

        log->counter--;

        for (int i = 0; i < max - 1; i++)  // Shift all missions to left

        {

            strcpy(log->missions[i], log->missions[i + 1]);

        }

        strcpy(log->missions[max - 1], mission);  // Add to last

        printf("%s logged (%d)\n", mission, log->counter);

        log->counter++;

    }

}

// Executing task 2

void rescue\_log()

{

    Log log;

    init\_log(&log);

    log\_mission(&log, "Mis1");

    log\_mission(&log, "Mis2");

    log\_mission(&log, "Mis3");

    log\_mission(&log, "Mis4");

    log\_mission(&log, "Mis5");

    log\_mission(&log, "Mis6");

    log\_mission(&log, "Mis7");

    log\_mission(&log, "Mis8");

    printf("\n");

    for (int i = 0; i < max; i++)

    {

        printf("%d. %s\n", i, log.missions[i]);

    }

    printf("\n");

}

// Task 3: Damaged robot tracker

typedef struct singly\_node {

    char damaged[name];

    struct singly\_node \*next;

} singly\_node;

typedef struct doubly\_node {

    char repaired[name];

    struct doubly\_node \*prev;

    struct doubly\_node \*next;

} doubly\_node;

// For damaged robots

singly\_node\* createsingly(char \*damaged)

{

    singly\_node \*new\_node = (singly\_node \*)malloc(sizeof(singly\_node));

    strcpy(new\_node->damaged, damaged);

    new\_node->next = NULL;

    return new\_node;

}

void insert\_damaged(singly\_node \*\*head, char \*damaged)

{

    singly\_node \*new\_node = createsingly(damaged);

    if (\*head == NULL)

    {

        \*head = new\_node;

    }

    else

    {

        singly\_node \*temp = \*head;

        while (temp->next != NULL)

        {

            temp = temp->next;

        }

        temp->next = new\_node;

    }

    printf("Damaged robot: %s added to repair list\n", damaged);

}

singly\_node\* remove\_damaged(singly\_node \*\*head, char \*damaged)

{

    if (\*head == NULL)

    {

        return NULL;

    }

    singly\_node \*current = \*head;

    singly\_node \*prev = NULL;

    // If head node has the robot to be removed

    if (strcmp(current->damaged, damaged) == 0)

    {

        \*head = current->next;

        printf("%s robot removed from damaged list\n", damaged);

        return current;

    }

    // If not, then searching

    while (current != NULL && strcmp(current->damaged, damaged) != 0)

    {

        prev = current;

        current = current->next;

    }

    // Even if not found

    if (current == NULL)

    {

        printf("%s robot not found in damaged list\n", damaged);

        return NULL;

    }

    prev->next = current->next;

    printf("%s robot removed from damaged list\n", damaged);

    return current;

}

// For repaired robots

doubly\_node\* create\_doubly(char \*repaired)

{

    doubly\_node \*new\_node = (doubly\_node \*)malloc(sizeof(doubly\_node));

    strcpy(new\_node->repaired, repaired);

    new\_node->prev = NULL;

    new\_node->next = NULL;

    return new\_node;

}

void insert\_repaired(doubly\_node \*\*head, char \*repaired)

{

    doubly\_node \*new\_node = create\_doubly(repaired);

    if (\*head == NULL)

    {

        \*head = new\_node;

    }

    else

    {

        doubly\_node \*temp = \*head;

        while (temp->next != NULL)

        {

            temp = temp->next;

        }

        temp->next = new\_node;

        new\_node->prev = temp;

    }

    printf("Repaired robot: %s added to inspection list\n", repaired);

}

void traverse\_forward(doubly\_node \*head)

{

    printf("Forward traversal: ");

    doubly\_node \*temp = head;

    while (temp != NULL)

    {

        printf("%s -> ", temp->repaired);

        temp = temp->next;

    }

    printf("NULL\n");

}

void traverse\_backward(doubly\_node \*head)

{

    if (head == NULL)

    {

        printf("Backward traversal: NULL\n");

        return;

    }

    doubly\_node \*temp = head; // First, find last node

    while (temp->next != NULL)

    {

        temp = temp->next;

    }

    printf("Backward traversal: ");

    while (temp != NULL)

    {

        printf("%s -> ", temp->repaired);

        temp = temp->prev;

    }

    printf("NULL\n");

}

// Executing task 3

void damaged\_robot\_tracker()

{

    singly\_node \*damaged\_robots = NULL;

    doubly\_node \*repaired\_robots = NULL;

    insert\_damaged(&damaged\_robots, "Digger");

    insert\_damaged(&damaged\_robots, "Drone");

    printf("\n");

    singly\_node \*repaired = remove\_damaged(&damaged\_robots, "Digger");

    if (repaired != NULL)

    {

        insert\_repaired(&repaired\_robots, repaired->damaged);

        free(repaired);  // Free node the transferring data

    }

    printf("\n");

    traverse\_forward(repaired\_robots);

    traverse\_backward(repaired\_robots);

    printf("\n");

}

// Task 4: Priority redeployment

typedef struct circular\_node {

    char urgent[name];

    struct circular\_node \*next;

} circular\_node;

circular\_node\* create\_circular(char \*urgent)

{

    circular\_node \*new\_node = (circular\_node \*)malloc(sizeof(circular\_node));

    strcpy(new\_node->urgent, urgent);

    new\_node->next = NULL;

    return new\_node;

}

void insert\_priority(circular\_node \*\*head, char \*urgent)

{

    circular\_node \*new\_node = create\_circular(urgent);

    if (\*head == NULL)

    {

        \*head = new\_node;

        new\_node->next = new\_node;  // Points to itself

    }

    else

    {

        circular\_node \*temp = \*head;

        while (temp->next != \*head)

        {

            temp = temp->next;

        }

        temp->next = new\_node;

        new\_node->next = \*head;

    }

    printf("%s robot added to priority redeployment list\n", urgent);

}

void traverse\_circular(circular\_node \*head, int rounds)

{

    if (head == NULL)

    {

        printf("Priority list is empty\n");

        return;

    }

    circular\_node \*temp = head;

    int nodes = 0;

    int count = 0;

    do  // Conuting number of nodes

    {

        nodes++;

        temp = temp->next;

    } while (temp != head);

    temp = head;

    printf("Traversing priority robots (%d rounds):\n", rounds);

    do

    {

        printf("Checking: %s\n", temp->urgent);

        temp = temp->next;

        count++;

        if (count % nodes == 0)  // Rounds completed

        {

            printf("Completed %d round(s)\n\n", (count / nodes));

        }

    } while (count < rounds \* nodes);

}

// Executing task 4

void priority\_redeployment()

{

    circular\_node \*priority\_robots = NULL;

    insert\_priority(&priority\_robots, "Scanner");

    insert\_priority(&priority\_robots, "Lift");

    printf("\n");

    traverse\_circular(priority\_robots, 2);

}

int main()

{

    mission\_and\_urgency();

    rescue\_log();

    damaged\_robot\_tracker();

    priority\_redeployment();

    return 0;

}

***SAMPLE OUTPUT:***

**PS C:\Users\AYUSH GUPTA\Desktop\c> gcc edsa.c**

**PS C:\Users\AYUSH GUPTA\Desktop\c> ./a**

**Scanner enqueued.**

**Digger enqueued.**

**Lift enqueued.**

**Light enqueued.**

**Drone enqueued.**

**Cutter enqueued.**

**Scanner pushed.**

**Digger pushed.**

**Lift pushed.**

**Light pushed.**

**Drone pushed.**

**Cutter pushed.**

**Executing urgent tasks: Cutter**

**Executing urgent tasks: Drone**

**Executing urgent tasks: Light**

**Executing urgent tasks: Lift**

**Executing urgent tasks: Digger**

**Executing urgent tasks: Scanner**

**Mis1 logged (0)**

**Mis2 logged (1)**

**Mis3 logged (2)**

**Mis4 logged (3)**

**Mis5 logged (4)**

**Mis6 logged (5)**

**Reporting oldest mission: Mis1**

**Mis7 logged (5)**

**Reporting oldest mission: Mis2**

**Mis8 logged (5)**

**0. Mis3**

**1. Mis4**

**2. Mis5**

**3. Mis6**

**4. Mis7**

**5. Mis8**

**Damaged robot: Digger added to repair list**

**Damaged robot: Drone added to repair list**

**Digger robot removed from damaged list**

**Repaired robot: Digger added to inspection list**

**Forward traversal: Digger -> NULL**

**Backward traversal: Digger -> NULL**

**Scanner robot added to priority redeployment list**

**Lift robot added to priority redeployment list**

**Traversing priority robots (2 rounds):**

**Checking: Scanner**

**Checking: Lift**

**Completed 1 round(s)**

**Checking: Scanner**

**Checking: Lift**

**Completed 2 round(s)**