ASSIGNMENT-3

NAME: KOWSHIK

ROLL NO: ME24I1027

DATE: 15/04/25

PROBLEM: Autonomous Crane Fabrication Unit

Scenario: You’re designing an Autonomous Crane Fabrication Unit for assembling crane parts (e.g., "Boom", "Counterweight", "Cable", "Hook", "Base") in a construction yard. The unit uses:

● Part Delivery System (Queue): Parts arrive via trucks and queue for fabrication.

● Crane Task Manager (Stack): Robots stack parts in LIFO order for load balancing.

● Assembly Storage Unit (Array): Finished cranes store in an array-based yard (size: 6 slots). If full, the oldest is deployed.

● Repair and Upgrade Tracker (Linked Lists):

○ Faulty cranes go to a singly linked list.

○ Repaired cranes move to a doubly linked list for dual checks.

○ High-priority cranes cycle in a circular linked list for urgent upgrades.

CODE:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Part a: Queue and Stack implementation for Part Delivery and Crane Assembly

#define MAX\_PARTS 6

// Queue implementation

typedef struct QueueNode

{

char part[20];

struct QueueNode\* next;

} QueueNode;

typedef struct

{

QueueNode\* front;

QueueNode\* rear;

int size;

} Queue;

void initQueue(Queue\* q)

{

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

q->size = 0;

}

void enqueue(Queue\* q, const char\* part)

{

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

strcpy(newNode->part, part);

newNode->next = NULL;

if (q->rear == NULL)

{

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

} else

{

q->rear->next = newNode;

q->rear = newNode;

}

q->size++;

}

char\* dequeue(Queue\* q)

{

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

QueueNode\* temp = q->front;

char\* part = strdup(temp->part); // Make a copy to return

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

if (q->front == NULL)

{

q->rear = NULL;

}

free(temp);

q->size--;

return part;

}

// Stack implementation

typedef struct StackNode

{

char part[20];

struct StackNode\* next;

} StackNode;

void push(StackNode\*\* top, const char\* part)

{

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

strcpy(newNode->part, part);

newNode->next = \*top;

\*top = newNode;

}

char\* pop(StackNode\*\* top)

{

if (\*top == NULL) return NULL;

StackNode\* temp = \*top;

char\* part = strdup(temp->part); // Make a copy to return

\*top = (\*top)->next;

free(temp);

return part;

}

// Part b: Array implementation for Assembly Storage Unit

#define STORAGE\_SIZE 6

char\* assemblyStorage[STORAGE\_SIZE];

int storageCount = 0;

void addCraneToStorage(const char\* craneName)

{

// If storage is full, remove the oldest (index 0)

if (storageCount == STORAGE\_SIZE)

{

printf("Storage full! Deploying oldest crane: %s\n", assemblyStorage[0]);

free(assemblyStorage[0]);

// Shift all cranes left

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

{

assemblyStorage[i] = assemblyStorage[i + 1];

}

storageCount--;

}

// Add new crane to the end

assemblyStorage[storageCount] = strdup(craneName);

storageCount++;

}

// Part c: Linked lists for Repair and Upgrade Tracker

// Singly linked list for faulty cranes

typedef struct FaultyNode

{

char crane[20];

struct FaultyNode\* next;

} FaultyNode;

// Doubly linked list for repaired cranes

typedef struct RepairedNode

{

char crane[20];

struct RepairedNode\* next;

struct RepairedNode\* prev;

} RepairedNode;

// Part d: Circular linked list for priority upgrades

typedef struct PriorityNode

{

char crane[20];

struct PriorityNode\* next;

} PriorityNode;

void addToFaultyList(FaultyNode\*\* head, const char\* crane)

{

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

strcpy(newNode->crane, crane);

newNode->next = \*head;

\*head = newNode;

}

FaultyNode\* removeFromFaultyList(FaultyNode\*\* head, const char\* crane)

{

FaultyNode \*current = \*head, \*prev = NULL;

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

{

prev = current;

current = current->next;

}

if (current == NULL) return NULL;

if (prev == NULL)

{

\*head = current->next;

} else {

prev->next = current->next;

}

return current;

}

void addToRepairedList(RepairedNode\*\* head, const char\* crane)

{

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

strcpy(newNode->crane, crane);

newNode->next = \*head;

newNode->prev = NULL;

if (\*head != NULL)

{

(\*head)->prev = newNode;

}

\*head = newNode;

}

void addToPriorityList(PriorityNode\*\* head, const char\* crane)

{

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

strcpy(newNode->crane, crane);

if (\*head == NULL)

{

newNode->next = newNode;

\*head = newNode;

} else {

PriorityNode\* temp = \*head;

while (temp->next != \*head)

{

temp = temp->next;

}

temp->next = newNode;

newNode->next = \*head;

}

}

void traversePriorityList(PriorityNode\* head, int times)

{

if (head == NULL) return;

PriorityNode\* temp = head;

printf("Priority Upgrade Traversal (%d times):\n", times);

for (int i = 0; i < times; i++)

{

do {

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

temp = temp->next;

} while (temp != head);

}

printf("(back to start)\n");

}

int main() {

printf("=== Autonomous Crane Fabrication Unit Simulation ===\n\n");

// Part a: Part Delivery and Crane Assembly

printf("Part a: Part Delivery and Crane Assembly\n");

Queue partQueue;

initQueue(&partQueue);

// Enqueue all 6 parts

enqueue(&partQueue, "Boom");

enqueue(&partQueue, "Counterweight");

enqueue(&partQueue, "Cable");

enqueue(&partQueue, "Hook");

enqueue(&partQueue, "Base");

enqueue(&partQueue, "Pulley");

printf("Parts enqueued in order: Boom, Counterweight, Cable, Hook, Base, Pulley\n");

// Dequeue and push onto stack

StackNode\* assemblyStack = NULL;

while (partQueue.size > 0)

{

char\* part = dequeue(&partQueue);

push(&assemblyStack, part);

free(part);

}

// Pop to show assembly order

printf("Assembly order (LIFO): ");

while (assemblyStack != NULL)

{

char\* part = pop(&assemblyStack);

printf("%s ", part);

free(part);

}

printf("\n");

// Creativity Bonus explanation

printf("Creativity Bonus: LIFO is suitable because the hook (last part added) needs to be assembled last for immediate lifting readiness, while the base (first part added) needs to be installed first for stability.\n\n");

// Part b: Assembly Storage Unit

printf("Part b: Assembly Storage Unit\n");

// Insert first 6 cranes

for (int i = 1; i <= 6; i++)

{

char craneName[10];

sprintf(craneName, "Crane%d", i);

addCraneToStorage(craneName);

}

printf("Storage after adding first 6 cranes:\n");

for (int i = 0; i < storageCount; i++)

{

printf("[%d] %s\n", i, assemblyStorage[i]);

}

// Handle overflow for Crane7 and Crane8

addCraneToStorage("Crane7");

addCraneToStorage("Crane8");

printf("\nStorage after adding 2 more cranes (total 8):\n");

for (int i = 0; i < storageCount; i++)

{

printf("[%d] %s\n", i, assemblyStorage[i]);

}

// Creativity Bonus explanation

printf("Creativity Bonus: The oldest crane is deployed when storage is full to prioritize immediate site demands, ensuring construction projects aren't delayed while maintaining a full inventory of ready cranes.\n\n");

// Part c: Faulty Crane Tracker

printf("Part c: Faulty Crane Tracker\n");

FaultyNode\* faultyList = NULL;

RepairedNode\* repairedList = NULL;

// Add Crane2 and Crane5 to faulty list

addToFaultyList(&faultyList, "Crane2");

addToFaultyList(&faultyList, "Crane5");

printf("Faulty cranes in singly linked list:\n");

FaultyNode\* temp = faultyList;

while (temp != NULL)

{

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

temp = temp->next;

}

printf("NULL\n");

// Move Crane2 to repaired list

FaultyNode\* repairedCrane = removeFromFaultyList(&faultyList, "Crane2");

if (repairedCrane != NULL)

{

addToRepairedList(&repairedList, repairedCrane->crane);

free(repairedCrane);

}

printf("\nAfter repairing Crane2:\n");

printf("Remaining faulty cranes: ");

temp = faultyList;

while (temp != NULL)

{

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

temp = temp->next;

}

printf("NULL\n");

printf("Repaired cranes (doubly linked list) forward traversal: ");

RepairedNode\* rTemp = repairedList;

while (rTemp != NULL)

{

printf("%s -> ", rTemp->crane);

rTemp = rTemp->next;

}

printf("NULL\n");

printf("Repaired cranes backward traversal: ");

rTemp = repairedList;

// Go to end

while (rTemp != NULL && rTemp->next != NULL)

{

rTemp = rTemp->next;

}

while (rTemp != NULL)

{

printf("%s -> ", rTemp->crane);

rTemp = rTemp->prev;

}

printf("NULL\n");

// Creativity Bonus explanation

printf("Creativity Bonus: Crane2's cable was frayed and detected by sensors. It was automatically replaced by repair bots with a new carbon-fiber cable, then moved to the doubly-linked list for dual verification of the repair.\n\n");

// Part d: Priority Upgrades

printf("Part d: Priority Upgrades\n");

PriorityNode\* priorityList = NULL;

// Add Crane1 and Crane4 to priority list

addToPriorityList(&priorityList, "Crane1");

addToPriorityList(&priorityList, "Crane4");

// Traverse twice

traversePriorityList(priorityList, 2);

// Creativity Bonus explanation

printf("Creativity Bonus: Crane4 is getting a magnetic grip upgrade for handling steel beams more efficiently, while Crane1 is receiving extended reach capabilities (from 50m to 75m) for high-rise construction projects.\n\n");

// Clean up memory

for (int i = 0; i < storageCount; i++)

{

free(assemblyStorage[i]);

}

// Free faulty list

while (faultyList != NULL)

{

FaultyNode\* toFree = faultyList;

faultyList = faultyList->next;

free(toFree);

}

// Free repaired list

while (repairedList != NULL)

{

RepairedNode\* toFree = repairedList;

repairedList = repairedList->next;

free(toFree);

}

// Free priority list

if (priorityList != NULL)

{

PriorityNode\* start = priorityList;

PriorityNode\* current = priorityList->next;

while (current != start)

{

PriorityNode\* toFree = current;

current = current->next;

free(toFree);

}

free(start);

}

return 0;

}

REPORT FOR THIS CODE:

**1. Problem Statement & Objectives**

**Problem Being Solved**

The construction industry requires efficient crane assembly, storage, and maintenance. Manual tracking of crane parts, faulty units, and upgrades is error-prone and slow. This project simulates an **Autonomous Crane Fabrication Unit** that automates:

* **Part delivery & assembly**
* **Storage management**
* **Fault detection & repair tracking**
* **Priority upgrades**

**Key Objectives**

1. **Simulate part delivery** using a **Queue** (FIFO).
2. **Assemble parts** using a **Stack** (LIFO) for load balancing.
3. **Manage crane storage** in a fixed-size **Array**, replacing the oldest crane when full.
4. **Track faulty cranes** in a **Singly Linked List** and **repaired cranes** in a **Doubly Linked List**.
5. **Handle priority upgrades** using a **Circular Linked List** for continuous processing.

**2. Design Explanation**

**Why These Data Structures?**

| **Component** | **Data Structure** | **Reason for Choice** |
| --- | --- | --- |
| Part Delivery | **Queue (FIFO)** | Parts arrive in order and should be processed sequentially. |
| Assembly Process | **Stack (LIFO)** | Ensures the last part (e.g., "Hook") is assembled last for immediate use. |
| Storage Management | **Array (Fixed Size)** | Provides O(1) access and easy overflow handling by removing the oldest crane. |
| Faulty Cranes | **Singly linked list** | Simple insertion/deletion of faulty cranes. |
| Repaired Cranes | **Doubly linked list** | Allows forward/backward traversal for dual verification. |
| Priority Upgrades | **Circular linked list** | Enables continuous processing of high-priority cranes. |

**Efficiency Justification**

* **Queue & Stack**: O(1) insertion/deletion.
* **Array**: O(1) access, O(n) shifting when full (acceptable for small storage).
* **Linked Lists**: O(1) insertion/deletion at head, O(n) traversal (efficient for dynamic repairs/upgrades).

**3. Logic of the Code**

**Step-by-Step Workflow**

1. **Part Delivery (Queue)**
   * Parts (Boom, Counterweight, Cable, Hook, Base, Pulley) are enqueued.
   * The robot dequeues each part and pushes it onto a **stack** (LIFO).
   * The stack determines assembly order (Pulley → Base → Hook → Cable → Counterweight → Boom).
2. **Storage Management (Array)**
   * Cranes (Crane1 to Crane6) fill the storage array.
   * Adding Crane7 and Crane8 removes Crane1 and Crane2 (oldest first).
3. **Faulty Crane Tracker (Linked Lists)**
   * Crane2 and Crane5 are marked faulty and added to a **singly linked list**.
   * After repair, Crane2 moves to a **doubly linked list** for dual verification.
4. **Priority Upgrades (Circular Linked List)**
   * Crane1 and Crane4 are added for urgent upgrades.
   * The circular list allows repeated traversal for priority handling.

**4. Variables & Functions Used**

**Key Variables**

| **Variable** | **Purpose** |
| --- | --- |
| Queue partQueue | Stores parts in FIFO order. |
| Stack assemblyStack | Reorders parts in LIFO for assembly. |
| assemblyStorage[6] | Fixed-size array for crane storage. |
| FaultyNode\* | Singly linked list for faulty cranes. |
| RepairedNode\* | Doubly linked list for repaired cranes. |
| PriorityNode\* | Circular linked list for upgrades. |

**Key Functions**

| **Function** | **Purpose** |
| --- | --- |
| enqueue() / dequeue() | Queue operations for part delivery. |
| push() / pop() | Stack operations for assembly. |
| addCraneToStorage() | Manages crane storage with overflow handling. |
| addToFaultyList() | Adds faulty cranes to a singly linked list. |
| addToRepairedList() | Moves repaired cranes to a doubly linked list. |
| addToPriorityList() | Handles upgrades in a circular list. |

**5. Output Screenshot (Example)**

*(Optional but useful for visualization)*

Copy

Download

=== Autonomous Crane Fabrication Unit Simulation ===

Part a: Part Delivery and Crane Assembly

Parts enqueued: Boom, Counterweight, Cable, Hook, Base, Pulley

Assembly order (LIFO): Pulley Base Hook Cable Counterweight Boom

Part b: Assembly Storage Unit

Storage after adding 6 cranes:

[0] Crane1, [1] Crane2, ..., [5] Crane6

Adding Crane7 → Deploys Crane1

Adding Crane8 → Deploys Crane2

Part c: Faulty Crane Tracker

Faulty cranes: Crane5 → NULL

Repaired cranes (forward): Crane2 → NULL

Repaired cranes (backward): Crane2 → NULL

Part d: Priority Upgrades

Priority Upgrade Traversal (2 times):

Crane1 → Crane4 → Crane1 → Crane4 → (back to start)

**6. Conclusion**

This project demonstrates how **data structures optimize crane fabrication**:  
✅ **Queue** ensures orderly part delivery.  
✅ **Stack** balances assembly workload.  
✅ **Array** efficiently manages limited storage.  
✅ **Linked lists** track repairs and upgrades dynamically.

**Future Improvements:**

* Use a **hash table** for faster faulty crane lookup.
* Implement **multithreading** for parallel processing.