DSLT ASSIGNMENT: Robotic Line simulator

K K Bharadwaj

ME24B1023

13th April, 2025

Objectives:

· To create a highly effective data management program to handle a considerable number of tasks, including edge cases in the realm of automotive production.

· To use several different implementations of data structures for each specific subsystem, including **Queues** *(for Part delivery system, queuing each part for processing)*, **Stacks** *(Used as a LIFO parts picker for assembly)*, **Arrays** *(Storage units for completed prototypes, shipping out the oldest version once storage limit is reached),* **Linked Lists** *(Singly LL used for defective protypes, Doubly LL for repaired, Circular LL for VIPs)*

Design explanation:

To explain the different design choices takes in the code, I shall divide this section into 2 parts i.e Why specific data structures and how they solve the problems efficiently

· The different data structures used and reasoning: As given in the question,

1) We first use Queues for the part delivery system. This is advantageous as we may simply "Funnel" parts as they are received from user input, with each part being presented in the precise order they were received in (i.e FIFO).

2) For the assembly line task management, we use Stack implementation. We use stacks simply due to ease of use, as we may pop elements (here, parts) as required for assembly

3) For storage of units, we use Arrays. Arrays are not dynamic memory types and thus we may set the maximum storage that may be used for protype car storage

4) Finally, we use Linked lists, within which we use Singly linked list for defective protypes, doubly linked lists for repaired protypes and for VIPs, we use circular linked list as they must be given precedence.

· How efficiency is achieved:

To understand how this allows for efficient function, we must understand the role of each data structure used in its implementation

1) Queue: As stated before, allows us to funnel parts in without significant changes

2) Stacks: Allows robot to simply take element of queue and place it within stacks. Immensely simplified data management as elements can be orderly accessed based on need

3) Arrays allow us to fix a size, allowing us to simply push out prototypes and based on requirement remove the oldest ones upon reaching the storage cap.

4) The several different types of linked list offer us advantages, not just limited to reducing time complexity, reducing redundant memory usage, allowing us to add priority to different functions and so on.

Code Logic:

The code logic shall be explained with the aid of screen shots.

main.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>


#define Size 10

typedef struct { //Queue implementation using stack
    char *data[Size];
    int frt;
    int rear;
    int size;
} Queue;

void Qchk(Queue *q) {
    q->frt = 0;
    q->rear = -1;
    q->size = 0;
}

int EmptChk(Queue *q) { //Checks if queue is empty
    return q->size == 0;
}

int Fullchk(Queue *q) { //Checks if queue is full by comparing with size
    return q->size == Size;
}

void enqueue(Queue *q, char *item) { //Function to enqueue
    if (Fullchk(q)) {
        printf("The queue is full!\n");
        return;
    }
    q->rear = (q->rear + 1) % Size; //Adds element to rear position
    q->data[q->rear] = item;
    q->size++; //increments size
}

char *dequeue(Queue *q) { //Function to dequeue
    if (EmptChk(q))
        return NULL;
    char *item = q->data[q->frt];
    q->frt = (q->frt + 1) % Size; //Increments front, i.e ignores first element
    q->size--; //decreases size
    return item;
}
```

First, we observe standard queue implementation, i.e we have Enqueue, Dequeue as checks to see if the queue is full or empty. Given user input, I have designed queue around it, however it is capable of handling any user input.

Next, we observe stack implementation. Here, all is normal, however,

```c

//Stack implementation (Robotic manager)
#define StkSize 10

typedef struct { //Stack implementaiton using structure
    char *data[StkSize];
    int top;
} Stack;

void stkin(Stack *s) {
    s->top = -1;
}

int EmptChkS(Stack *s) {
    return s->top == -1;
}

int FullChkS(Stack *s) {
    return s->top == StkSize - 1;
}

void push(Stack *s, char *item) { //Function to push value (here, item) to stack
    if (FullChkS(s)) {
        printf("Stack overflow\n");
        return;
    }
    s->data[++s->top] = item; //increments top
}

char *pop(Stack *s) { //function to pop topmost element
    if (EmptChkS(s))
        return NULL;
    return s->data[s->top--];//decreases top
}

void partDel() {
    printf("<-------------- Part Delivery Simulation ------------------>\n");

    // Initialize queue with arriving parts
```

```
    printf("<--------------- Part Delivery Simulation ------------------->\n");

    // Initialize queue with arriving parts
    Queue partQueue;
    Qchk(&partQueue);
    char *parts[] = {"Engine", "Chassis", "Wheels", "Doors", "Battery", "Hood"}; //As given by given question, can be replaces by user input if required
    int partCount = sizeof(parts) / sizeof(parts[0]);

    for (int i = 0; i < partCount; i++) {
        enqueue(&partQueue, parts[i]);
        printf("The '%s' has arrived on the conveyor belt.\n", parts[i]);
    }
    printf("\n");
    //LIFO implementation of robot arm picking up parts
    Stack robotArmStack;
    stkin(&robotArmStack);

    while (!EmptChk(&partQueue)) {
        char *part = dequeue(&partQueue);
        push(&robotArmStack, part);
        printf("Robot arm picked up '%s' and placed it in its stack.\n", part);
    }

    // Pop the stack to reveal the final assembly order
    printf("\nAssembly Order (Last In, First Out):\n");
    while (!EmptChkS(&robotArmStack)) {
        char *assembled = pop(&robotArmStack);
        printf("Placing, assembling: %s\n", assembled);
        printf("Assembled\n");
    }


    printf("\nNote: The robot arm ensures the \"Hood\" is installed last as required\n\n");
    printf("<---------------------Prototype completed-------------------->\n\n");
}
```

We observe that for Part simulation, the code follows the following logic

→Initialize Queue, user input added to queue element by element

→ Acknowledge addition of elements to queue

→ Use LIFO Implementation (stack) to simulate robot picking parts of the conveyor belt and placing in stack

→ The picking up of parts involves dequeuing of elements and pushing to stack

->Elements are then popped from stack to show storage order

```c
//Array implementation for storage
void Garage() {
    printf("<------Car Storage--------->\n");

    #define StorageCap 8
    char *garage[StorageCap];
    int count = 0;

    for (int i = 1; i <= 10; i++) { //Adds cars for max 10, however storage cap 8
        char Buffer[10];
        sprintf(Buffer, "Car%d", i);
        // Duplicate the string so memory persists.
        char *carPrototype = strdup(Buffer);

        if (count < StorageCap) {
            garage[count++] = carPrototype;
            printf("%s stored in the garage.\n", carPrototype);
        } else {
            //If Garage full, ships out oldest protype
            printf("Garage full! Shipping out the oldest prototype: %s.\n", garage[0]);
            printf("Shipped!\n");
            free(garage[0]);
            for (int j = 1; j < count; j++) { //Shifts all protypes once left to reinitiate oldest protype
                garage[j - 1] = garage[j];
            }
            garage[count - 1] = carPrototype;
            printf("%s stored in the garage.\n", carPrototype);
        }
    }

    // Display final garage occupancy.
    printf("\nFinal garage occupancy: \n");
    for (int i = 0; i < count; i++) {
        printf("%s \n ", garage[i]);
    }
    // Free allocated memory.
    for (int i = 0; i < count; i++) {
        free(garage[i]); //Optimization
    }
    printf("\n<-------------------------Prototype handling-------------------->\n\n");
}
```

Here, array implementation follows the following steps:

→ Prototypes completed of the line are designated as Car 1, Car 2, Car 3 and so on

→ Here, Garage is our array, elements are added to array from carPrototype

->iterates through using count++

->Once full, ships out first element (which will be oldest prototype)

→ Shifts all elements back to make previous second element as current oldest prototype

->Displays final garage occupancy by iterating through array and printing it’s elements

```c
    printf("\n<------------------------Prototype handling-------------------->\n\n");
}
//Linked list implementation
// Singly Linked List Node for Defective Prototypes
typedef struct SNode {
    char *data;
    struct SNode *next;
} SNode;

SNode* createSNode(char *data) { //Node creation
    SNode *node = (SNode *)malloc(sizeof(SNode));
    node->data = strdup(data);
    node->next = NULL;
    return node;
}

void SLLin(SNode **head, char *data) { //Function to insert prototype into LL
    SNode *new_node = createSNode(data);
    new_node->next = *head;
    *head = new_node;
}

int SLLdel(SNode **head, char *data) {
    SNode *current = *head, *prev = NULL;

    while (current != NULL) {
        if (strcmp(current->data, data) == 0) {
            if (prev == NULL) {  // removing head
                *head = current->next;
            } else {
                prev->next = current->next;
            }
            free(current->data);
            free(current);
            return 1;  // Successfully removed.
        }
        prev = current;
        current = current->next;
    }
    return 0; //Edge case
}

void RecSLL(SNode *head) {
    while (head != NULL) {
        printf("%s\n", head->data);
        head = head->next;
    }
}
```

Singly linked list implementation is standard, where each node refers to a defunct prototype that is categorized.

```c
// Doubly Linked List Node for Repaired Prototypes
typedef struct DNode {
    char *data;
    struct DNode *next;
    struct DNode *prev;
} DNode;

DNode* createDNode(char *data) { //Node creation
    DNode *node = (DNode *)malloc(sizeof(DNode));
    node->data = strdup(data);
    node->next = NULL;
    node->prev = NULL;
    return node;
}

typedef struct {
    DNode *head;
    DNode *tail;
} DoublyLL;

void initDLL(DoublyLL *list) {
    list->head = NULL;
    list->tail = NULL;
}

void insertDLL(DoublyLL *list, char *data) {
    DNode *new_node = createDNode(data);
    if (list->head == NULL) {
        list->head = list->tail = new_node;
    } else {
        list->tail->next = new_node;
        new_node->prev = list->tail;
        list->tail = new_node;
    }
}

void ForwardDLL(DoublyLL *list) {
    DNode *current = list->head;
    while (current != NULL) {
        printf("%s\n", current->data);
        current = current->next;
    }
}

void BackDLL(DoublyLL *list) {
    DNode *current = list->tail;
    while (current != NULL) {
        printf("%s\n", current->data);
        current = current->prev;
    }
}
```

Standard doubly linked list implementation

```c
258
259 void DefectiveTracking() {
260     printf("<-------Defective Prototypes------>\n");
261
262     SNode *defectiveList = NULL;
263     SLLin(&defectiveList, "Car3, sending to repair shop"); //Assuming cars 3 and 6 are defective
264     SLLin(&defectiveList, "Car6, sending to repair shop");
265
266     printf("Defective Prototypes (Singly Linked List):\n");
267     RecSLL(defectiveList);
268
269     // After repair, remove "Car3" from the defective list and insert it into a doubly linked list.
270     if (SLLdel(&defectiveList, "Car3")) {
271         DoublyLL repairedList;
272         initDLL(&repairedList);
273         insertDLL(&repairedList, "Car3");
274         printf("\nCar3 has been repaired and moved to the Doubly Linked List for further inspection.\n");
275
276         printf("\nTraversing repaired prototypes (forward):\n");
277         ForwardDLL(&repairedList);
278         printf("Traversing repaired prototypes (backward):\n");
279         BackDLL(&repairedList);
280
281         // Free nodes in the doubly linked list.
282         DNode *curr = repairedList.head;
283         while (curr != NULL) {
284             DNode *next = curr->next;
285             free(curr->data);
286             free(curr);
287             curr = next;
288         }
289     } else {
290         printf("\nCar3 was not found in the defective list, repaired!\n");
291     }
292
293     // Free the remaining nodes of the defective singly linked list.
294     SNode *curr = defectiveList;
295     while (curr != NULL) {
296         SNode *next = curr->next;
297         free(curr->data);
298         free(curr);
299         curr = next;
300     }
301
302     printf("\nDetail: Car3 had loose shock mounts; a precise robotic wrench tightened them to meet quality standards, we apologize for
        the delay.\n\n");
303 }
```

Here, the function defective tracking follows the following logic:

→ Assuming cars 3 and 6 are defective, defective-list (Here an SLL) is used to categorize and store

→ Once stored, we pass them to Doubly linked list (Repaired list) to then display as repaired (consider lines 270 to 274)

→ We traverse through the prototypes present within the DLL using Forward DLL and backward DLL.

→ To optimize memory managment, we use free to then delete the remaining cars within as they are no longer needed within (Case in point, line 290, showing car3 after repairs has been shipped out)

```c
// Circular Linked List Node for VIP prototypes.
typedef struct CNode {
    char *data;
    struct CNode *next;
} CNode;

typedef struct {
    CNode *head;
} CircularLL;

void initCLL(CircularLL *list) {
    list->head = NULL;
}

void insertCircular(CircularLL *list, char *data) { //Function to create node
    CNode *new_node = (CNode *)malloc(sizeof(CNode));
    new_node->data = strdup(data);
    if (list->head == NULL) {
        list->head = new_node;
        new_node->next = list->head;
    } else {
        CNode *current = list->head; //Keeps track of head
        while (current->next != list->head)
            current = current->next;
        current->next = new_node;
        new_node->next = list->head;
    }
}

void TraverseCLL(CircularLL *list, int rounds) {
    if (list->head == NULL)
        return;
    CNode *current = list->head;
    for (int r = 0; r < rounds; r++) {
        printf("--- Upgrade Round %d ---\n", r + 1);
        do {
            printf("%s\n", current->data);
            current = current->next;
        } while (current != list->head);
    }
}
```

Standard CLL implementation ,however we use traverse CLL in only one way. We also note that we include a count (upgrade rounds) for each exclusive car

```c
}

void MemCLL(CircularLL *list) {
    if (list->head == NULL)
        return;
    CNode *start = list->head;
    CNode *current = start->next; //Frees data as required
    while (current != start) {
        CNode *next = current->next;
        free(current->data);
        free(current);
        current = next;
    } //I implemented this function as a quick access way to free data as required
    free(start->data);
    free(start);
    list->head = NULL; //By doing this, I may call upon free mememory without rewriting
}

void VIPS() { //VIP implementation
    printf("<------------------VIP Upgrades!--------------------->\n");

    CircularLL vipList;
    initCLL(&vipList);

    // Add VIP prototypes.
    insertCircular(&vipList, "Car1");
    insertCircular(&vipList, "Car5");
    printf("VIP prototypes recieving upgrades: Car1 and Car5.\n");

    // Simulate two rounds of upgrade checks.
    TraverseCLL(&vipList, 2);
    printf("\nUpgrade1: Car5 now features a Fusion reactor, allowing it to run nearly indefinetly without refueling\n\n");
    printf("\nUpgrade2: Car 1 now features magnetic suspension, allowing active ride changing dynamics for a luxurious ride! "); //My own addition
    MemCLL(&vipList);
}
//Main function
int main() {
    partDel();
    Garage();
    DefectiveTracking();
    VIPS();
    return 0;
}
```

Within memory CLL, we perform two actions, once perform linking and freeing memory

Logic followed in VIPS is as follows:

→ We obtain user input cars to perform upgrades for, then add to Viplist (CLL)

→ After storage and “alteration”, we remove the cars use MemCLL to free memory and print the “upgrades” that we had done to the cars

This is the logic of the overall code

Sample output:

Output

Clear

```
<--------------- Part Delivery Simulation ------------------->
The 'Engine' has arrived on the conveyor belt.
The 'Chassis' has arrived on the conveyor belt.
The 'Wheels' has arrived on the conveyor belt.
The 'Doors' has arrived on the conveyor belt.
The 'Battery' has arrived on the conveyor belt.
The 'Hood' has arrived on the conveyor belt.

Robot arm picked up 'Engine' and placed it in its stack.
Robot arm picked up 'Chassis' and placed it in its stack.
Robot arm picked up 'Wheels' and placed it in its stack.
Robot arm picked up 'Doors' and placed it in its stack.
Robot arm picked up 'Battery' and placed it in its stack.
Robot arm picked up 'Hood' and placed it in its stack.

Assembly Order (Last In, First Out):
Placing, assembling: Hood
Assembled
Placing, assembling: Battery
Assembled
Placing, assembling: Doors
Assembled
Placing, assembling: Wheels
Assembled
Placing, assembling: Chassis
Assembled
Placing, assembling: Engine
Assembled

Note: The robot arm ensures the "Hood" is installed last as required

<----------------------Prototype completed-------------------->

<------Car Storage--------->
Car1 stored in the garage.
Car2 stored in the garage.
Car3 stored in the garage.
Car4 stored in the garage.
Car5 stored in the garage.
Car6 stored in the garage.
Car7 stored in the garage.
Car8 stored in the garage.
Garage full! Shipping out the oldest prototype: Car1.
Shipped!
Car9 stored in the garage.
Garage full! Shipping out the oldest prototype: Car2.
Shipped!
Car10 stored in the garage.

Final garage occupancy:
Car3
 Car4
 Car5
 Car6
 Car7
 Car8
 Car9
 Car10

<-------------------------Prototype handling-------------------->

<-------Defective Prototypes------>
Defective Prototypes (Singly Linked List):
Car6, sending to repair shop
Car3, sending to repair shop

Car3 was not found in the defective list, repaired!

Detail: Car3 had loose shock mounts; a precise robotic wrench tightened them to meet quality standards, we apologize for the delay.

<------------------VIP Upgrades!--------------------->
VIP prototypes recieving upgrades: Car1 and Car5.
--- Upgrade Round 1 ---
Car1
Car5
--- Upgrade Round 2 ---
Car1
Car5

Upgrade1: Car5 now features a Fusion reactor, allowing it to run nearly indefinetly without refueling
```

Fin