Day16_Assignment_Abhirami

1.

```c
/*create a node in a  linked list which will have the following details of student
1. Name, roll number, class, section, an array having marks of any three subjects
Create a linked list for 5 students and print it.*/

#include <stdio.h>
#include <stdlib.h>

typedef struct student{
        char name[50];
        int rollNumber;
        int class;
        char section[5];
        int marks[3];
        struct student *next;
}student;

student* createNode();

int main() {
        student *first = NULL;
        for (int i = 0; i < 5; i++) {
```

```
        printf("\nEnter student %d details:", i+1);
        student *newNode = createNode();
        if (first == NULL) {
        first = newNode;
        } else {
        student *temp = first;
        while (temp->next!= NULL) {
        temp = temp->next;
        }
        temp->next = newNode;
        }
        }
        student *temp = first;
        printf("\n---------- Student Details ----------\n\n");
        printf("%-15s %-10s %-8s %-10s %-20s\n", "Name", "Roll No.", "Class", "Section", "Marks");
        printf("-----------------------------------------------------\n");

        while (temp != NULL) {
        printf("%-15s %-10d %-8d %-10s ", temp->name, temp->rollNumber, temp->class,
temp->section);
        for (int i = 0; i < 3; i++) {
        printf("%d ", temp->marks[i]);
        }
        printf("\n");
        temp = temp->next;
```

```
}



        while (first!= NULL) {
        student *temp = first;
        first = first->next;
        free(temp);
        }
        return 0;
}


student* createNode() {
        student *newNode = (student*)malloc(sizeof(student));
        if (!newNode) {
        printf("Memory allocation failed!\n");
        exit(1);
        }


        printf("\nEnter Name: ");
        scanf("%s", newNode->name);
        printf("Enter Roll Number: ");
        scanf("%d", &newNode->rollNumber);
        printf("Enter Class: ");
        scanf("%d", &newNode->class);
        printf("Enter Section: ");
```

```
        scanf("%s", newNode->section);
        printf("Enter Marks of 3 subjects: ");
        for (int i = 0; i < 3; i++) {
        scanf("%d", &newNode->marks[i]);
        }
        newNode->next = NULL;
        return newNode;
}
```

2.

```
/*Implementation of adding nodes to a linked list*/
#include <stdio.h>
#include <stdlib.h>

typedef struct node {
        int data;
        struct node *next;
} Node;

void InsertFront(Node **, int);
void InsertMiddle(Node *, int, int);
void InsertEnd(Node **, int);
void printList(Node *);

int main() {
```

```
    Node *head = NULL;


    InsertEnd(&head, 6);
    InsertEnd(&head, 8);
    InsertEnd(&head, 10);
    InsertFront(&head, 4);
    InsertFront(&head, 0);
    InsertMiddle(head, 2, 7);  // Inserts 7 after position 2 (1-based indexing)


    printList(head);
    return 0;
}


void InsertEnd(Node **ptrHead, int nData) {
    Node *newNode = (Node *)malloc(sizeof(Node));
    if (newNode == NULL) {
    printf("Memory allocation failed.\n");
    return;
    }
    newNode->data = nData;
    newNode->next = NULL;


    if (*ptrHead == NULL) {
    *ptrHead = newNode;
    } else {
```

```
        Node *ptrTail = *ptrHead;
        while (ptrTail->next != NULL) {
        ptrTail = ptrTail->next;
        }
        ptrTail->next = newNode;
        }
}

void InsertFront(Node **ptrHead, int nData) {
        Node *newNode = (Node *)malloc(sizeof(Node));
        if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
        }
        newNode->data = nData;
        newNode->next = *ptrHead;
        *ptrHead = newNode;
}

void InsertMiddle(Node *ptrHead, int after, int nData) {
        if (ptrHead == NULL) {
        printf("The list is empty. Cannot insert at position %d.\n", after);
        return;
        }
```

```
        Node *newNode = (Node *)malloc(sizeof(Node));
        if (newNode == NULL) {
        printf("Memory allocation failed.\n");
        return;
        }
        newNode->data = nData;
        newNode->next = NULL;

        Node *ptrCurrent = ptrHead;
        int count = 1;

        while (ptrCurrent != NULL && count < after) {
        ptrCurrent = ptrCurrent->next;
        count++;
        }

        if (ptrCurrent == NULL) {
        printf("Invalid position: List has fewer than %d nodes.\n", after);
        free(newNode);
        return;
        }

        newNode->next = ptrCurrent->next;
        ptrCurrent->next = newNode;
}
```

```
void printList(Node *node) {
        while (node != NULL) {
        printf("%d -> ", node->data);
        node = node->next;
        }
        printf("NULL\n");
}
```

3.

```
/*Problem 1: Reverse a Linked List
Write a C program to reverse a singly linked list. The program should traverse the list, reverse the pointers between the nodes, and display the reversed list.
Requirements:
Define a function to reverse the linked list iteratively.
Update the head pointer to the new first node.
Display the reversed list.

Example Input:
rust
Copy code
Initial list: 10 -> 20 -> 30 -> 40
Example Output:
rust
Copy code
```

```c
Reversed list: 40 -> 30 -> 20 -> 10*/


#include <stdio.h>

#include <stdlib.h>


typedef struct node{

        int data;

        struct node *next;

}Node;


void InsertEnd(Node **, int);

void ReverseList(Node **);

void printList(Node *);


int main() {

        Node *head = NULL;


        InsertEnd(&head, 10);

        InsertEnd(&head, 20);

        InsertEnd(&head, 30);

        InsertEnd(&head, 40);


        printf("Initial list: ");

        printList(head);
```

```
	ReverseList(&head);


	printf("\nReversed list: ");
	printList(head);


	return 0;
}


void InsertEnd(Node **ptrHead, int nData) {
	// Creating a node
	Node *newNode = (Node*)malloc(sizeof(Node));
	newNode->data = nData;
	newNode->next = NULL;


	// If the linked list is empty, make ptrHead point to the new node created
	if (*ptrHead == NULL) {
	*ptrHead = newNode;
	} else {
	// Traverse till the last node and insert the new node at the end
	Node *ptrTail = *ptrHead;  // Start at the head
	while (ptrTail->next != NULL) {  // Traverse to the last node
	ptrTail = ptrTail->next;
	}
	ptrTail->next = newNode;  // Insert the new node at the end
	}
```

```
}

void ReverseList(Node **ptrHead) {
        Node *prev = NULL;
        Node *current = *ptrHead;
        Node *nextNode;

        //1. Traverse the list and reverse the pointers
        while (current!= NULL) {
        nextNode = current->next;
        current->next = prev;
        prev = current;
        current = nextNode;
        }

        //2. Update the head pointer to the new first node
        *ptrHead = prev;
}

void printList(Node *node) {
        while (node != NULL) {
        printf(" %d->", node->data);
        node = node->next;
        }
        printf("\n");
```

```
}
```

4.

```
/*Problem 2: Find the Middle Node

Write a C program to find and display the middle node of a singly linked list. If the list has an even number of nodes, display the first middle node.

Requirements:

Use two pointers: one moving one step and the other moving two steps.

When the faster pointer reaches the end, the slower pointer will point to the middle node.

Example Input:

rust

Copy code

List: 10 -> 20 -> 30 -> 40 -> 50

Example Output:

scss

Copy code

Middle node: 30*/


#include <stdio.h>

#include <stdlib.h>


typedef struct node{

        int data;

        struct node *next;

}Node;
```

```c
void InsertEnd(Node **, int);
void findMiddle(Node **);
void printList(Node *);

int main() {
	Node *head = NULL;

	InsertEnd(&head, 10);
	InsertEnd(&head, 20);
	InsertEnd(&head, 30);
	InsertEnd(&head, 40);
	InsertEnd(&head, 50);

	printf("List: ");
	printList(head);

	findMiddle(&head);

	return 0;
}

void InsertEnd(Node **ptrHead, int nData) {
	// Creating a node
	Node *newNode = (Node*)malloc(sizeof(Node));
```

```
        newNode->data = nData;
        newNode->next = NULL;

        // If the linked list is empty, make ptrHead point to the new node created
        if (*ptrHead == NULL) {
        *ptrHead = newNode;
        } else {
        // Traverse till the last node and insert the new node at the end
        Node *ptrTail = *ptrHead;  // Start at the head
        while (ptrTail->next != NULL) {  // Traverse to the last node
        ptrTail = ptrTail->next;
        }
        ptrTail->next = newNode;  // Insert the new node at the end
        }
}

void findMiddle(Node **head) {
        Node *slowPtr = *head;
        Node *fastPtr = *head;
        while(fastPtr->next!=NULL) {
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next->next;
        }
        printf("Middle node: %d\n", slowPtr->data);
}
```

```
void printList(Node *node) {
        while (node != NULL) {
        printf(" %d->", node->data);
        node = node->next;
        }
        printf("\n");
}
```

5.

```
/*Problem 3: Detect and Remove a Cycle in a Linked List

Write a C program to detect if a cycle (loop) exists in a singly linked list and remove it if present. Use Floyd’s Cycle Detection Algorithm (slow and fast pointers) to detect the cycle.

Requirements:

Detect the cycle in the list.

If a cycle exists, find the starting node of the cycle and break the loop.

Display the updated list.

Example Input:

rust

Copy code

List: 10 -> 20 -> 30 -> 40 -> 50 -> (points back to 30)

Example Output:

rust

Copy code

Cycle detected and removed.
```

```c
Updated list: 10 -> 20 -> 30 -> 40 -> 50*/


#include <stdio.h>

#include <stdlib.h>


// Node structure

typedef struct node {

        int data;

        struct node *next;

} Node;


// Function Prototypes

void InsertEnd(Node **, int);

int findCycle(Node **);

void createCycle(Node **, int);

void printList(Node *);


// Main Function

int main() {

        Node *head = NULL;

        int n, data, cycleIndex;


        // Input number of nodes

        printf("Enter the number of elements in the linked list: ");

        scanf("%d", &n);
```

```
        // Input elements
        for (int i = 0; i < n; i++) {
        printf("Enter element %d: ", i + 1);
        scanf("%d", &data);
        InsertEnd(&head, data);
        }

        // Display initial list
        printf("Initial list: ");
        printList(head);

        // Ask user whether to create a cycle
        printf("Do you want to create a cycle? (Enter -1 for no cycle or index of node [0-%d] to point the last node): ", n - 1);
        scanf("%d", &cycleIndex);

        // Create cycle if user specifies
        if (cycleIndex >= 0 && cycleIndex < n) {
        createCycle(&head, cycleIndex);
        }

        // Detect and remove cycle
        if (findCycle(&head)) {
        printf("Cycle detected and removed.\n");
```

```
        } else {
        printf("No cycle detected.\n");
        }

        // Print updated list
        printf("Updated list: ");
        printList(head);

        return 0;
}

// Function to insert a node at the end of the list
void InsertEnd(Node **ptrHead, int nData) {
        Node *newNode = (Node *)malloc(sizeof(Node));
        newNode->data = nData;
        newNode->next = NULL;

        if (*ptrHead == NULL) {
        *ptrHead = newNode;
        } else {
        Node *ptrTail = *ptrHead;
        while (ptrTail->next != NULL) {
        ptrTail = ptrTail->next;
        }
        ptrTail->next = newNode;
```

```
        }
}

// Function to create a cycle at the specified index
void createCycle(Node **ptrHead, int index) {
        Node *cycleNode = NULL, *tail = *ptrHead;
        int count = 0;

        // Traverse the list to find the node at the given index
        while (tail->next != NULL) {
        if (count == index) {
        cycleNode = tail;
        }
        tail = tail->next;
        count++;
        }

        // Create the cycle
        if (cycleNode != NULL) {
        tail->next = cycleNode;
        printf("Cycle created: last node points to node with data %d.\n", cycleNode->data);
        }
}

// Function to detect and remove a cycle in the linked list
```

```
int findCycle(Node **ptrHead) {
        if (*ptrHead == NULL || (*ptrHead)->next == NULL) {
        return 0; // No cycle possible in empty or single-node list
        }

        Node *slowPtr = *ptrHead;
        Node *fastPtr = *ptrHead;

        // Step 1: Detect the cycle using Floyd's Algorithm
        while (fastPtr != NULL && fastPtr->next != NULL) {
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next->next;

        if (slowPtr == fastPtr) {
        // Step 2: Find the start of the cycle
        slowPtr = *ptrHead;
        Node *prev = NULL; // Keep track of the last node in the cycle

        while (slowPtr != fastPtr) {
        prev = fastPtr;
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next;
        }

        // Step 3: Remove the cycle
```

```
            prev->next = NULL;
            return 1; // Cycle detected and removed
            }
            }
            return 0; // No cycle detected
}

// Function to print the linked list
void printList(Node *node) {
            while (node != NULL) {
            printf("%d", node->data);
            if (node->next != NULL) {
            printf(" -> ");
            }
            node = node->next;
            }
            printf("\n");
}
```