## 1) Find Middle element of a singly linked list in one pass.

WTD: Use two pointers, one moving twice as fast as the other, to find the middle element in a single pass.

(e.g.: I/P: 1->2->3->4->5; O/P: 3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to find the middle element of the linked list
int findMiddle(struct Node* head) {
    struct Node* slow_ptr = head;
    struct Node* fast_ptr = head;

    if (head != NULL) {
        while (fast_ptr != NULL && fast_ptr->next != NULL) {
            fast_ptr = fast_ptr->next->next; // Move fast pointer by two
steps
            slow_ptr = slow_ptr->next;       // Move slow pointer by one
step
        }
        return slow_ptr->data; // The slow pointer is now at the middle
element
    }
    return -1; // Return -1 if the list is empty
}

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
```

```c
    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

// Function to display the linked list
void display(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
    insert(&head, 4);
    insert(&head, 5);

    printf("Linked List: ");
    display(head);

    int middle = findMiddle(head);
    if (middle != -1) {
        printf("Middle element: %d\n", middle);
    } else {
        printf("The list is empty.\n");
    }
```

```
    return 0;
}
```

## 2) Find the length of a singly linked list.

WTD: Traverse the list from head to tail, incrementing a counter to find its length.
(e.g.: I/P: 1->2->3->4; O/P: 4)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to find the length of the linked list
int findLength(struct Node* head) {
    int length = 0;
    struct Node* current = head;

    while (current != NULL) {
        length++;
        current = current->next;
    }

    return length;
}

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
```

```c
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

// Function to display the linked list
void display(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
    insert(&head, 4);

    printf("Linked List: ");
    display(head);

    int length = findLength(head);
    printf("Length of the linked list: %d\n", length);

    return 0;
}
```

## 3) Reverse a linked list.

WTD: Traverse the list while reversing the next pointers of each node.
(e.g.: I/P: 1->2->3; O/P: 3->2->1)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the beginning of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = *head;
    *head = new_node;
}

// Function to reverse the linked list
void reverse(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    *head = prev; // Update the head to the new first node (previously the last node)
}

// Function to display the linked list
void display(struct Node* head) {
```

```c
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);

    printf("Original Linked List: ");
    display(head);

    // Reverse the linked list
    reverse(&head);

    printf("Reversed Linked List: ");
    display(head);

    return 0;
}
```

## 4) Reverse a singly linked list without recursion.

WTD: Use an iterative method to reverse the next pointers of each node.
(e.g.: I/P: 1->2->3; O/P: 3->2->1)

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

// Define a singly linked list node structure
struct Node {
    int data;
```

```
    struct Node* next;
};

// Function to insert a new node at the beginning of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = *head;
    *head = new_node;
}

// Function to reverse the linked list iteratively
void reverse(struct Node** head) {
    struct Node* prev = NULL;
    struct Node* current = *head;
    struct Node* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }

    *head = prev; // Update the head to the new first node (previously the
last node)
}

// Function to display the linked list
void display(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;
```

```
    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);

    printf("Original Linked List: ");
    display(head);

    // Reverse the linked list
    reverse(&head);

    printf("Reversed Linked List: ");
    display(head);

    return 0;
}
```

## 5) Remove duplicate nodes in an unsorted linked list.

WTD: Use a hash table to record the occurrence of each node while traversing the list to remove duplicates.
(e.g.: I/P: 1->2->2->3; O/P: 1->2->3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
```

```c
        *head = new_node;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_node;
    }
}

// Function to remove duplicate nodes from the linked list
void removeDuplicates(struct Node* head) {
    if (head == NULL) {
        return;
    }

    // Create a hash table to keep track of encountered values
    int hashTable[1000] = {0}; // Assuming a maximum of 1000 unique values

    struct Node* current = head;
    struct Node* prev = NULL;

    while (current != NULL) {
        if (hashTable[current->data] == 0) {
            // This is the first occurrence of the current data
            hashTable[current->data] = 1;
            prev = current;
        } else {
            // Duplicate node, remove it
            prev->next = current->next;
            free(current);
            current = prev;
        }
        current = current->next;
    }
}

// Function to display the linked list
void display(struct Node* head) {
    struct Node* current = head;
```

```
    while (current != NULL) {
        printf("%d", current->data);
        if (current->next != NULL) {
            printf(" -> ");
        }
        current = current->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 2);
    insert(&head, 3);

    printf("Original Linked List: ");
    display(head);

    // Remove duplicate nodes
    removeDuplicates(head);

    printf("Linked List after Removing Duplicates: ");
    display(head);

    return 0;
}
```

## 6) Find the nth node from the end of a singly linked list.

WTD: Use two pointers, move one n nodes ahead, then move both until the first one reaches the end.
(e.g.: I/P: 1->2->3->4 (n=2); O/P: 3)

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

```c
// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_node;
    }
}

// Function to find the nth node from the end of the linked list
struct Node* findNthFromEnd(struct Node* head, int n) {
    if (head == NULL || n <= 0) {
        return NULL;
    }

    struct Node* firstPtr = head;
    struct Node* secondPtr = head;

    // Move the first pointer n nodes ahead
    for (int i = 0; i < n; i++) {
        if (firstPtr == NULL) {
            return NULL; // List length is less than n
        }
        firstPtr = firstPtr->next;
    }
```

```c
    // Move both pointers until the first pointer reaches the end
    while (firstPtr != NULL) {
        firstPtr = firstPtr->next;
        secondPtr = secondPtr->next;
    }

    return secondPtr;
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
    insert(&head, 4);

    printf("Original Linked List: 1 -> 2 -> 3 -> 4\n");

    int n = 2; // Find the 2nd node from the end
    struct Node* nthNode = findNthFromEnd(head, n);

    if (nthNode != NULL) {
        printf("The %dth node from the end is: %d\n", n, nthNode->data);
    } else {
        printf("The list is too short or n is invalid.\n");
    }

    return 0;
}
```

## 7) Move the last element to the front of a given linked list.

WTD: Find the last node and its previous node, change their pointers to move the last node to the front.
(e.g.: I/P: 1->2->3->4; O/P: 4->1->2->3)

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

```c
// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_node;
    }
}

// Function to move the last element to the front of the linked list
void moveLastToFront(struct Node** head) {
    if (*head == NULL || (*head)->next == NULL) {
        return; // No or only one element in the list
    }

    struct Node* current = *head;
    struct Node* previous = NULL;

    // Traverse to the last node and its previous node
    while (current->next != NULL) {
        previous = current;
        current = current->next;
    }

    // Move the last node to the front
```

```c
    previous->next = NULL;
    current->next = *head;
    *head = current;
}

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

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
    insert(&head, 4);

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

    moveLastToFront(&head);

    printf("Linked List after moving last element to the front: ");
    printList(head);

    return 0;
}
```

## 8) Delete alternate nodes of a linked list.

WTD: Traverse the list and remove every alternate node.
(e.g.: I/P: 1->2->3->4; O/P: 1->3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_node;
    }
}

// Function to delete alternate nodes of the linked list
void deleteAlternateNodes(struct Node* head) {
    if (head == NULL || head->next == NULL) {
        return; // No or only one element in the list
    }

    struct Node* current = head;
    struct Node* temp;
```

```c
    while (current != NULL && current->next != NULL) {
        temp = current->next;
        current->next = temp->next;
        free(temp);
        current = current->next;
    }
}

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

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
    insert(&head, 4);

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

    deleteAlternateNodes(head);

    printf("Linked List after deleting alternate nodes: ");
    printList(head);

    return 0;
}
```

## 9) Pairwise swap elements of a linked list.

WTD: Swap every two adjacent nodes by adjusting their pointers.
(e.g.: I/P: 1->2->3->4; O/P: 2->1->4->3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_node;
    }
}

// Function to pairwise swap elements of the linked list
void pairwiseSwap(struct Node** head) {
    if (*head == NULL || (*head)->next == NULL) {
        return; // No or only one element in the list
    }

    struct Node* prev = NULL;
    struct Node* current = *head;
```

```c
    while (current != NULL && current->next != NULL) {
        struct Node* nextNode = current->next;
        current->next = nextNode->next;
        nextNode->next = current;

        if (prev != NULL) {
            prev->next = nextNode;
        } else {
            *head = nextNode; // Update the head if swapping the first two
nodes
        }

        prev = current;
        current = current->next;
    }
}

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

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
    insert(&head, 4);

    printf("Original Linked List: ");
    printList(head);
```

```c
    pairwiseSwap(&head);

    printf("Linked List after pairwise swapping: ");
    printList(head);

    return 0;
}
```

## 10) Check if a given linked list contains a cycle and what would be the starting node?

WTD: Use Floyd's cycle-finding algorithm to detect the cycle and then find its starting node.
(e.g.: I/P: 1->2->3 (3 points back to 1); O/P: True)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insert(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = new_node;
    }
```

```
}

// Function to detect a cycle and find its starting node
bool detectAndFindCycle(struct Node* head, struct Node** startingNode) {
    struct Node* slowPtr = head;
    struct Node* fastPtr = head;

    // Detect a cycle using Floyd's cycle-finding algorithm
    while (fastPtr != NULL && fastPtr->next != NULL) {
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next->next;

        if (slowPtr == fastPtr) {
            break; // Cycle detected
        }
    }

    // If there is no cycle, return false
    if (fastPtr == NULL || fastPtr->next == NULL) {
        return false;
    }

    // Find the starting node of the cycle
    slowPtr = head;
    while (slowPtr != fastPtr) {
        slowPtr = slowPtr->next;
        fastPtr = fastPtr->next;
    }

    *startingNode = slowPtr;
    return true;
}

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insert(&head, 1);
    insert(&head, 2);
    insert(&head, 3);
```

```
    // Create a cycle (3 points back to 1)
    head->next->next->next = head;

    struct Node* startingNode = NULL;
    bool hasCycle = detectAndFindCycle(head, &startingNode);

    if (hasCycle) {
        printf("The linked list contains a cycle.\n");
        printf("Starting Node of the Cycle: %d\n", startingNode->data);
    } else {
        printf("The linked list does not contain a cycle.\n");
    }

    return 0;
}
```

## 11) Intersection point of two linked lists.

WTD: Use two pointers, one for each list, and traverse to find the intersection point.
(e.g.: I/P: 1->2->3 & 4->5->3; O/P: 3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to find the length of a linked list
int length(struct Node* head) {
    int count = 0;
    struct Node* current = head;
    while (current != NULL) {
        count++;
        current = current->next;
    }
```

```
    return count;
}

// Function to find the intersection point of two linked lists
struct Node* findIntersection(struct Node* head1, struct Node* head2) {
    int len1 = length(head1);
    int len2 = length(head2);

    struct Node* ptr1 = head1;
    struct Node* ptr2 = head2;

    // Move the longer list's pointer ahead by the difference in lengths
    if (len1 > len2) {
        int diff = len1 - len2;
        for (int i = 0; i < diff; i++) {
            ptr1 = ptr1->next;
        }
    } else if (len2 > len1) {
        int diff = len2 - len1;
        for (int i = 0; i < diff; i++) {
            ptr2 = ptr2->next;
        }
    }

    // Move both pointers simultaneously until they meet at the
intersection point
    while (ptr1 != NULL && ptr2 != NULL) {
        if (ptr1 == ptr2) {
            return ptr1; // Intersection point found
        }
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    return NULL; // No intersection point found
}

int main() {
    struct Node* head1 = NULL;
    struct Node* head2 = NULL;
```

```c
// Create the first linked list: 1->2->3
struct Node* node1 = (struct Node*)malloc(sizeof(struct Node));
node1->data = 1;
node1->next = NULL;
head1 = node1;

struct Node* node2 = (struct Node*)malloc(sizeof(struct Node));
node2->data = 2;
node2->next = NULL;
node1->next = node2;

struct Node* node3 = (struct Node*)malloc(sizeof(struct Node));
node3->data = 3;
node3->next = NULL;
node2->next = node3;

// Create the second linked list: 4->5
struct Node* node4 = (struct Node*)malloc(sizeof(struct Node));
node4->data = 4;
node4->next = NULL;
head2 = node4;

struct Node* node5 = (struct Node*)malloc(sizeof(struct Node));
node5->data = 5;
node5->next = NULL;
node4->next = node5;

// Connect the two linked lists at node3
node5->next = node3;

// Find the intersection point
struct Node* intersection = findIntersection(head1, head2);

if (intersection != NULL) {
    printf("Intersection Point: %d\n", intersection->data);
} else {
    printf("No intersection point found.\n");
}
```

```c
    return 0;
}
```

## 12) Segregate even and odd nodes in a linked list.

WTD: Use two pointers to rearrange nodes such that all even and odd elements are together. (e.g.: I/P: 1->2->3->4; O/P: 2->4->1->3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

// Function to segregate even and odd nodes in a linked list
struct Node* segregateEvenOdd(struct Node* head) {
    if (head == NULL || head->next == NULL) {
        return head;
    }
```

```c
    struct Node* evenStart = NULL;
    struct Node* evenEnd = NULL;
    struct Node* oddStart = NULL;
    struct Node* oddEnd = NULL;

    struct Node* current = head;

    while (current != NULL) {
        int data = current->data;

        if (data % 2 == 0) { // Even node
            if (evenStart == NULL) {
                evenStart = current;
                evenEnd = current;
            } else {
                evenEnd->next = current;
                evenEnd = current;
            }
        } else { // Odd node
            if (oddStart == NULL) {
                oddStart = current;
                oddEnd = current;
            } else {
                oddEnd->next = current;
                oddEnd = current;
            }
        }

        current = current->next;
    }

    if (evenStart == NULL) {
        return oddStart;
    } else {
        evenEnd->next = oddStart;
        oddEnd->next = NULL;
        return evenStart;
    }
}
```

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

int main() {
    struct Node* head = NULL;

    // Insert elements into the linked list
    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    insertAtEnd(&head, 4);
    insertAtEnd(&head, 5);

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

    // Segregate even and odd nodes
    head = segregateEvenOdd(head);

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

    return 0;
}
```

## 13) Merge two sorted linked lists.

WTD: Use a temporary dummy node to hold the sorted list, compare each node and attach the smaller one to the dummy.
(e.g.: I/P: 1->3->5 & 2->4->6; O/P: 1->2->3->4->5->6)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the end of the linked list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* current = *head;
        while (current->next != NULL) {
            current = current->next;
        }
        current->next = newNode;
    }
}

// Function to merge two sorted linked lists
struct Node* mergeSortedLists(struct Node* list1, struct Node* list2) {
    struct Node dummy;
    dummy.next = NULL;
    struct Node* tail = &dummy;

    while (1) {
```

```c
        if (list1 == NULL) {
            tail->next = list2;
            break;
        } else if (list2 == NULL) {
            tail->next = list1;
            break;
        }

        if (list1->data <= list2->data) {
            tail->next = list1;
            list1 = list1->next;
        } else {
            tail->next = list2;
            list2 = list2->next;
        }
        tail = tail->next;
    }

    return dummy.next;
}

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

int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;

    // Insert elements into the first sorted linked list
    insertAtEnd(&list1, 1);
    insertAtEnd(&list1, 3);
    insertAtEnd(&list1, 5);
```

```
    // Insert elements into the second sorted linked list
    insertAtEnd(&list2, 2);
    insertAtEnd(&list2, 4);
    insertAtEnd(&list2, 6);

    printf("First Sorted Linked List: ");
    printList(list1);

    printf("Second Sorted Linked List: ");
    printList(list2);

    // Merge the two sorted linked lists
    struct Node* mergedList = mergeSortedLists(list1, list2);

    printf("Merged Sorted Linked List: ");
    printList(mergedList);

    return 0;
}
```

## 14) Add two numbers represented by linked lists.

WTD: Traverse both lists, sum the corresponding nodes, and manage the carry.
(e.g.: I/P: 2->4 & 5->6 (24 + 56); O/P: 8->0)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to insert a new node at the beginning of the linked list
void insertAtBegin(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
```

```c
    *head = newNode;
}

// Function to add two numbers represented by linked lists
struct Node* addLists(struct Node* list1, struct Node* list2) {
    struct Node* result = NULL;
    struct Node* current = NULL;
    int carry = 0;

    while (list1 != NULL || list2 != NULL) {
        int sum = carry;

        if (list1 != NULL) {
            sum += list1->data;
            list1 = list1->next;
        }

        if (list2 != NULL) {
            sum += list2->data;
            list2 = list2->next;
        }

        carry = sum / 10;
        sum %= 10;

        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = sum;
        newNode->next = NULL;

        if (result == NULL) {
            result = newNode;
            current = newNode;
        } else {
            current->next = newNode;
            current = newNode;
        }
    }

    if (carry > 0) {
        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
```

```c
        newNode->data = carry;
        newNode->next = NULL;
        current->next = newNode;
    }

    return result;
}

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

int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;

    // Insert elements into the first linked list (represents 24)
    insertAtBegin(&list1, 4);
    insertAtBegin(&list1, 2);

    // Insert elements into the second linked list (represents 56)
    insertAtBegin(&list2, 6);
    insertAtBegin(&list2, 5);

    printf("First Linked List: ");
    printList(list1);

    printf("Second Linked List: ");
    printList(list2);

    // Add the two linked lists
    struct Node* sumList = addLists(list1, list2);

    printf("Sum Linked List: ");
```

```
    printList(sumList);

    return 0;
}
```

## 15) Find sum of two linked lists using stack.

WTD: Use two stacks to hold the numbers from each list, then pop and sum them, storing the result in a new list.
(e.g.: I/P: 2->4 & 5->6 (24 + 56); O/P: 8->0)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to push a digit onto a stack
void push(struct Node** top, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *top;
    *top = newNode;
}

// Function to pop a digit from a stack
int pop(struct Node** top) {
    if (*top == NULL) {
        return 0; // Stack is empty
    }
    struct Node* temp = *top;
    *top = (*top)->next;
    int data = temp->data;
    free(temp);
    return data;
}
```

```c
// Function to add two linked lists represented by stacks
struct Node* addLists(struct Node* stack1, struct Node* stack2) {
    struct Node* result = NULL;
    int carry = 0;

    while (stack1 != NULL || stack2 != NULL || carry != 0) {
        int num1 = pop(&stack1);
        int num2 = pop(&stack2);

        int sum = num1 + num2 + carry;
        carry = sum / 10;
        int digit = sum % 10;

        struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
        newNode->data = digit;
        newNode->next = result;
        result = newNode;
    }

    return result;
}

// Function to print a linked list
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%d -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* stack1 = NULL;
    struct Node* stack2 = NULL;

    // Push elements onto the first stack (represents 24)
    push(&stack1, 4);
    push(&stack1, 2);
```

```
    // Push elements onto the second stack (represents 56)
    push(&stack2, 6);
    push(&stack2, 5);

    printf("First Stack: ");
    printList(stack1);

    printf("Second Stack: ");
    printList(stack2);

    // Add the two stacks and store the result in a new stack
    struct Node* sumStack = addLists(stack1, stack2);

    printf("Sum Stack: ");
    printList(sumStack);

    return 0;
}
```

## 16) Compare two strings represented as linked lists.

WTD: Traverse both lists, comparing each node's value. If they are equal throughout, the lists are equal.
(e.g.: I/P: 'a'->'b'->'c' & 'a'->'b'->'c'; O/P: Equal)

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

// Define a singly linked list node structure for characters
struct Node {
    char data;
    struct Node* next;
};

// Function to compare two linked lists representing strings
int compareStrings(struct Node* list1, struct Node* list2) {
    while (list1 != NULL && list2 != NULL) {
        if (list1->data != list2->data) {
            return 0; // Not equal
        }
```

```c
        list1 = list1->next;
        list2 = list2->next;
    }

    // If both lists have reached the end, they are equal
    return (list1 == NULL && list2 == NULL);
}

// Function to print a linked list of characters
void printList(struct Node* head) {
    struct Node* current = head;
    while (current != NULL) {
        printf("%c -> ", current->data);
        current = current->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* list1 = NULL;
    struct Node* list2 = NULL;

    // Create the first linked list representing "abc"
    struct Node* node1 = (struct Node*)malloc(sizeof(struct Node));
    node1->data = 'a';
    node1->next = NULL;
    list1 = node1;

    struct Node* node2 = (struct Node*)malloc(sizeof(struct Node));
    node2->data = 'b';
    node2->next = NULL;
    node1->next = node2;

    struct Node* node3 = (struct Node*)malloc(sizeof(struct Node));
    node3->data = 'c';
    node3->next = NULL;
    node2->next = node3;

    // Create the second linked list representing "abc"
    struct Node* node4 = (struct Node*)malloc(sizeof(struct Node));
```

```
    node4->data = 'a';
    node4->next = NULL;
    list2 = node4;

    struct Node* node5 = (struct Node*)malloc(sizeof(struct Node));
    node5->data = 'b';
    node5->next = NULL;
    node4->next = node5;

    struct Node* node6 = (struct Node*)malloc(sizeof(struct Node));
    node6->data = 'c';
    node6->next = NULL;
    node5->next = node6;

    printf("List 1: ");
    printList(list1);

    printf("List 2: ");
    printList(list2);

    // Compare the two lists
    int result = compareStrings(list1, list2);

    if (result) {
        printf("Equal\n");
    } else {
        printf("Not Equal\n");
    }

    return 0;
}
```

## 17) Clone a linked list with the next and random pointer.

WTD: Create a deep copy of the linked list including the random pointers using a hash table to map original nodes to their copies.
(e.g.: I/P: 1->2->3 (random pointers set randomly); O/P: Cloned list with same structure and random pointers)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
    struct Node* random;
};

// Function to insert a new node at the beginning of the list
void insertAtBegin(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
    newNode->random = NULL;
    *head = newNode;
}

// Function to clone a linked list with next and random pointers
struct Node* cloneLinkedList(struct Node* head) {
    if (head == NULL) {
        return NULL;
    }

    // Create a mapping of original nodes to their corresponding copies
    struct Node* originalToCopy[1000] = {NULL};

    // Create a new head for the cloned list
    struct Node* newHead = (struct Node*)malloc(sizeof(struct Node));
    newHead->data = head->data;
    newHead->next = NULL;
    newHead->random = NULL;

    originalToCopy[(unsigned long)head] = newHead;

    struct Node* currentOriginal = head;
    struct Node* currentNew = newHead;

    while (currentOriginal != NULL) {
```

```c
        // Clone the next pointer
        if (currentOriginal->next != NULL) {
            if (originalToCopy[(unsigned long)currentOriginal->next] ==
NULL) {
                originalToCopy[(unsigned long)currentOriginal->next] =
(struct Node*)malloc(sizeof(struct Node));
                originalToCopy[(unsigned long)currentOriginal->next]->data
= currentOriginal->next->data;
                originalToCopy[(unsigned long)currentOriginal->next]->next
= NULL;
                originalToCopy[(unsigned
long)currentOriginal->next]->random = NULL;
            }
            currentNew->next = originalToCopy[(unsigned
long)currentOriginal->next];
        }

        // Clone the random pointer
        if (currentOriginal->random != NULL) {
            if (originalToCopy[(unsigned long)currentOriginal->random] ==
NULL) {
                originalToCopy[(unsigned long)currentOriginal->random] =
(struct Node*)malloc(sizeof(struct Node));
                originalToCopy[(unsigned
long)currentOriginal->random]->data = currentOriginal->random->data;
                originalToCopy[(unsigned
long)currentOriginal->random]->next = NULL;
                originalToCopy[(unsigned
long)currentOriginal->random]->random = NULL;
            }
            currentNew->random = originalToCopy[(unsigned
long)currentOriginal->random];
        }

        currentOriginal = currentOriginal->next;
        currentNew = currentNew->next;
    }

    return newHead;
}
```

```c
// Function to print the linked list with both next and random pointers
void printLinkedList(struct Node* head) {
    while (head != NULL) {
        printf("Data: %d, ", head->data);

        if (head->random != NULL) {
            printf("Random: %d", head->random->data);
        }

        printf("\n");
        head = head->next;
    }
}

int main() {
    // Create the original linked list with next and random pointers
    struct Node* head = NULL;
    insertAtBegin(&head, 3);
    insertAtBegin(&head, 2);
    insertAtBegin(&head, 1);

    // Set random pointers
    head->random = head->next->next;
    head->next->random = head;
    head->next->next->random = head->next;

    printf("Original List:\n");
    printLinkedList(head);

    // Clone the linked list
    struct Node* clonedHead = cloneLinkedList(head);

    printf("\nCloned List:\n");
    printLinkedList(clonedHead);

    return 0;
}
```

## 18) Merge sort on a linked list.

WTD: Implement the Merge Sort algorithm on a linked list, splitting the list into halves and merging them back in sorted order.
(e.g.: I/P: 3->1->2; O/P: 1->2->3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to merge two sorted linked lists
struct Node* merge(struct Node* left, struct Node* right) {
    struct Node dummy;
    struct Node* tail = &dummy;
    dummy.next = NULL;

    while (left != NULL && right != NULL) {
        if (left->data < right->data) {
            tail->next = left;
            left = left->next;
        } else {
            tail->next = right;
            right = right->next;
        }
        tail = tail->next;
    }

    if (left != NULL) {
        tail->next = left;
    } else {
        tail->next = right;
    }
```

```c
    return dummy.next;
}

// Function to perform Merge Sort on a linked list
struct Node* mergeSort(struct Node* head) {
    if (head == NULL || head->next == NULL) {
        return head; // Base case: List is empty or has one element
    }

    struct Node* slow = head;
    struct Node* fast = head->next;

    // Use slow and fast pointers to split the list into two halves
    while (fast != NULL) {
        fast = fast->next;
        if (fast != NULL) {
            slow = slow->next;
            fast = fast->next;
        }
    }

    struct Node* left = head;
    struct Node* right = slow->next;
    slow->next = NULL; // Split the list into two halves

    left = mergeSort(left);   // Recursively sort the left half
    right = mergeSort(right); // Recursively sort the right half

    // Merge the sorted halves
    return merge(left, right);
}

// Function to insert a new node at the beginning of the list
void insertAtBegin(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = *head;
    *head = newNode;
}
```

```c
// Function to print a linked list
void printLinkedList(struct Node* head) {
    while (head != NULL) {
        printf("%d ", head->data);
        head = head->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;
    insertAtBegin(&head, 2);
    insertAtBegin(&head, 1);
    insertAtBegin(&head, 3);

    printf("Original List: ");
    printLinkedList(head);

    head = mergeSort(head);

    printf("Sorted List: ");
    printLinkedList(head);

    return 0;
}
```

## 19) Detect and remove loops in a linked list.

WTD: Use Floyd's algorithm to detect the loop and then remove it by setting the next pointer of the last node in the loop to NULL.
(e.g.: I/P: 1->2->3 (3 points back to 1); O/P: 1->2->3)

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

// Define a singly linked list node structure
struct Node {
    int data;
    struct Node* next;
};
```

```c
// Function to detect and remove a loop in a linked list
void detectAndRemoveLoop(struct Node* head) {
    struct Node* slow = head;
    struct Node* fast = head;
    struct Node* loopStart = NULL;

    // Detect the loop using Floyd's algorithm
    while (slow && fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if (slow == fast) {
            loopStart = slow;
            break;
        }
    }

    // If a loop is found, remove it
    if (loopStart) {
        struct Node* ptr1 = head;
        struct Node* ptr2 = loopStart;

        // Move one pointer to the head of the linked list
        while (ptr1->next != ptr2->next) {
            ptr1 = ptr1->next;
            ptr2 = ptr2->next;
        }

        // Find the last node in the loop
        while (ptr2->next != loopStart) {
            ptr2 = ptr2->next;
        }

        // Remove the loop by setting the next pointer of the last node to
NULL
        ptr2->next = NULL;
    }
}

// Function to insert a new node at the end of the linked list
```

```c
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;

    if (*head == NULL) {
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
}

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

int main() {
    struct Node* head = NULL;

    // Create a linked list with a loop (1->2->3->4->5->3)
    for (int i = 1; i <= 5; i++) {
        insertAtEnd(&head, i);
    }
    head->next->next->next->next->next = head->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing the loop: ");
    printLinkedList(head);

    return 0;
```

```
}
```

## 20) Flatten a multi-level linked list.

WTD: Use a stack or recursion to flatten the list so that all nodes are at the same level.
(e.g.: I/P: 1->2->3 (2 has child 4->5); O/P: 1->2->4->5->3)

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

// Define the Node structure
struct Node {
    int data;
    struct Node* next;
    struct Node* child;
};

// Function to append a new node at the end of a linked list
void append(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
    new_node->child = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

// Function to flatten a multi-level linked list
void flatten(struct Node* head) {
    if (head == NULL) {
        return;
```

```c
    }

    struct Node* current = head;
    while (current != NULL) {
        if (current->child != NULL) {
            struct Node* next = current->next;
            current->next = current->child;
            current->child = NULL;

            struct Node* temp = current->next;
            while (temp->next != NULL) {
                temp = temp->next;
            }
            temp->next = next;
        }
        current = current->next;
    }
}

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

int main() {
    struct Node* head = NULL;

    // Construct the multi-level linked list
    append(&head, 1);
    append(&head, 2);
    append(&head, 3);

    head->child = (struct Node*)malloc(sizeof(struct Node));
    head->child->data = 4;

    head->child->next = (struct Node*)malloc(sizeof(struct Node));
```

```
    head->child->next->data = 5;

    // Print the original multi-level linked list
    printf("Original Multi-level Linked List:\n");
    printList(head);

    // Flatten the linked list
    flatten(head);

    // Print the flattened linked list
    printf("\nFlattened Linked List:\n");
    printList(head);

    return 0;
}
```

## 21) Partition a linked list around a given value.

WTD: Traverse the linked list, creating two separate lists - one for values less than the partition value and another for values greater than or equal to the partition value. Finally, merge these lists.

(e.g.: I/P: 1->4->3->2->5->2, Partition Value: 3; O/P: 1->2->2->4->3->5)

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

// Define the Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to append a new node at the end of a linked list
void append(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;
```

```
    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

// Function to partition a linked list around a given value
struct Node* partition(struct Node* head, int partitionValue) {
    if (head == NULL || head->next == NULL) {
        return head;
    }

    struct Node* lessList = NULL;
    struct Node* lessTail = NULL;
    struct Node* greaterList = NULL;
    struct Node* greaterTail = NULL;
    struct Node* current = head;

    while (current != NULL) {
        if (current->data < partitionValue) {
            if (lessList == NULL) {
                lessList = current;
                lessTail = current;
            } else {
                lessTail->next = current;
                lessTail = current;
            }
        } else {
            if (greaterList == NULL) {
                greaterList = current;
                greaterTail = current;
            } else {
                greaterTail->next = current;
                greaterTail = current;
```

```c
            }
        }
        current = current->next;
    }

    if (lessList == NULL) {
        return greaterList;
    }

    lessTail->next = greaterList;
    if (greaterTail != NULL) {
        greaterTail->next = NULL;
    }

    return lessList;
}

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

int main() {
    struct Node* head = NULL;

    // Construct the linked list
    append(&head, 1);
    append(&head, 4);
    append(&head, 3);
    append(&head, 2);
    append(&head, 5);
    append(&head, 2);

    // Print the original linked list
    printf("Original Linked List:\n");
    printList(head);
```

```
    int partitionValue = 3;

    // Partition the linked list
    head = partition(head, partitionValue);

    // Print the partitioned linked list
    printf("\nPartitioned Linked List (around %d):\n", partitionValue);
    printList(head);

    return 0;
}
```

## 22) Remove all nodes in a linked list that have a specific value.

WTD: Traverse the linked list and remove any node that has a value matching the specified value. Make sure to properly update the next pointers and free any removed nodes.
(e.g.: I/P: 1->2->6->3->4->5->6, Value to Remove: 6; O/P: 1->2->3->4->5)

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

// Define the Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to append a new node at the end of a linked list
void append(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* temp = *head;
```

```c
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

// Function to remove nodes with a specific value from a linked list
void removeNodesWithValue(struct Node** head, int value) {
    if (*head == NULL) {
        return;
    }

    // Handle cases where the first node(s) have the specified value
    while (*head != NULL && (*head)->data == value) {
        struct Node* temp = *head;
        *head = (*head)->next;
        free(temp);
    }

    if (*head == NULL) {
        return;
    }

    struct Node* current = *head;

    while (current->next != NULL) {
        if (current->next->data == value) {
            struct Node* temp = current->next;
            current->next = temp->next;
            free(temp);
        } else {
            current = current->next;
        }
    }
}

// Function to print a linked list
void printList(struct Node* head) {
    while (head != NULL) {
```

```c
        printf("%d -> ", head->data);
        head = head->next;
    }
    printf("NULL\n");
}

int main() {
    struct Node* head = NULL;

    // Construct the linked list
    append(&head, 1);
    append(&head, 2);
    append(&head, 6);
    append(&head, 3);
    append(&head, 4);
    append(&head, 5);
    append(&head, 6);

    // Print the original linked list
    printf("Original Linked List:\n");
    printList(head);

    int valueToRemove = 6;

    // Remove nodes with the specified value
    removeNodesWithValue(&head, valueToRemove);

    // Print the modified linked list
    printf("\nLinked List after Removing Nodes with Value %d:\n",
valueToRemove);
    printList(head);

    return 0;
}
```

## 23) Convert a binary number represented by a linked list to an integer.

WTD: Traverse the linked list and convert the binary number represented by the linked list nodes to an integer. Use bit manipulation for the conversion.
(e.g.: I/P: 1->0->1; O/P: 5)

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

// Define the Node structure
struct Node {
    int data;
    struct Node* next;
};

// Function to append a new node at the end of a linked list
void append(struct Node** head, int data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = data;
    new_node->next = NULL;

    if (*head == NULL) {
        *head = new_node;
    } else {
        struct Node* temp = *head;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}

// Function to convert a binary linked list to an integer
int convertBinaryToInteger(struct Node* head) {
    int result = 0;

    while (head != NULL) {
        result = (result << 1) | head->data;
        head = head->next;
```

```
    }

    return result;
}

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

int main() {
    struct Node* head = NULL;

    // Construct the binary linked list (LSB to MSB)
    append(&head, 1);
    append(&head, 0);
    append(&head, 1);

    // Print the binary linked list
    printf("Binary Linked List:\n");
    printList(head);

    // Convert the binary linked list to an integer
    int result = convertBinaryToInteger(head);

    // Print the integer result
    printf("\nInteger Value: %d\n", result);

    return 0;
}
```

## 24) Find the common ancestor of two nodes in a binary tree represented as a doubly linked list.

WTD: Traverse the binary tree represented as a doubly linked list and find the common ancestor of the given two nodes. Utilize parent pointers in the doubly linked list to backtrack.
(e.g.: I/P: Nodes 6 and 9 in Binary Tree 4->5->6->7->8->9; O/P: 7)

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

// Define the structure for a node in the binary tree with parent pointers
struct TreeNode {
    int val;
    struct TreeNode* left;
    struct TreeNode* right;
    struct TreeNode* parent;
};

// Function to find the common ancestor of two nodes in a binary tree
struct TreeNode* findCommonAncestor(struct TreeNode* node1, struct
TreeNode* node2) {
    // Create a set to store ancestors
    struct TreeNode* ancestors[100];
    int top = -1;

    // Traverse from node1 to the root and add all ancestors to the set
    while (node1 != NULL) {
        ancestors[++top] = node1;
        node1 = node1->parent;
    }

    // Traverse from node2 to the root and check for common ancestors
    while (node2 != NULL) {
        for (int i = 0; i <= top; i++) {
            if (node2 == ancestors[i]) {
                return node2; // Common ancestor found
            }
        }
        node2 = node2->parent;
```

```
    }

    return NULL; // No common ancestor found
}

int main() {
    // Construct a binary tree as a doubly linked list with parent
pointers
    struct TreeNode* root = (struct TreeNode*)malloc(sizeof(struct
TreeNode));
    root->val = 4;
    root->left = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    root->left->val = 5;
    root->left->parent = root;
    root->left->right = (struct TreeNode*)malloc(sizeof(struct TreeNode));
    root->left->right->val = 6;
    root->left->right->parent = root->left;
    root->left->right->right = (struct TreeNode*)malloc(sizeof(struct
TreeNode));
    root->left->right->right->val = 7;
    root->left->right->right->parent = root->left->right;
    root->left->right->right->right = (struct
TreeNode*)malloc(sizeof(struct TreeNode));
    root->left->right->right->right->val = 8;
    root->left->right->right->right->parent = root->left->right->right;
    root->left->right->right->right->right = (struct
TreeNode*)malloc(sizeof(struct TreeNode));
    root->left->right->right->right->right->val = 9;
    root->left->right->right->right->right->parent =
root->left->right->right->right;

    // Nodes for which we want to find the common ancestor
    struct TreeNode* node1 = root->left->right->right; // Node with value
7
    struct TreeNode* node2 = root->left->right->right->right->right; //
Node with value 9

    // Find the common ancestor
    struct TreeNode* commonAncestor = findCommonAncestor(node1, node2);
```

```c
    if (commonAncestor != NULL) {
        printf("Common Ancestor: %d\n", commonAncestor->val);
    } else {
        printf("No Common Ancestor found.\n");
    }

    // Clean up the dynamically allocated memory (not shown in the
example)

    return 0;
}
```

## 25) Determine if a linked list is a palindrome.

WTD: Use a slow and fast pointer to find the middle of the list. Reverse the second half and compare it with the first half to determine if the linked list is a palindrome.
(e.g.: I/P: 1->2->2->1; O/P: True)

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

// Define the structure for a node in the linked list
struct ListNode {
    int val;
    struct ListNode* next;
};

// Function to reverse a linked list
struct ListNode* reverseList(struct ListNode* head) {
    struct ListNode* prev = NULL;
    struct ListNode* current = head;
    struct ListNode* next = NULL;

    while (current != NULL) {
        next = current->next;
        current->next = prev;
        prev = current;
        current = next;
```

```
    }

    return prev;
}

// Function to check if a linked list is a palindrome
bool isPalindrome(struct ListNode* head) {
    if (head == NULL || head->next == NULL) {
        return true; // Empty list or single-node list is a palindrome
    }

    struct ListNode* slow = head;
    struct ListNode* fast = head;

    // Move slow to the middle and fast to the end of the list
    while (fast->next != NULL && fast->next->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Reverse the second half of the list
    struct ListNode* secondHalf = reverseList(slow->next);

    // Compare the first half and the reversed second half
    struct ListNode* firstHalf = head;
    while (secondHalf != NULL) {
        if (firstHalf->val != secondHalf->val) {
            return false; // Not a palindrome
        }
        firstHalf = firstHalf->next;
        secondHalf = secondHalf->next;
    }

    return true; // It's a palindrome
}

int main() {
    // Construct a sample linked list
    struct ListNode* head = (struct ListNode*)malloc(sizeof(struct
ListNode));
```

```
    head->val = 1;
    head->next = (struct ListNode*)malloc(sizeof(struct ListNode));
    head->next->val = 2;
    head->next->next = (struct ListNode*)malloc(sizeof(struct ListNode));
    head->next->next->val = 2;
    head->next->next->next = (struct ListNode*)malloc(sizeof(struct
ListNode));
    head->next->next->next->val = 1;
    head->next->next->next->next = NULL;

    // Check if the linked list is a palindrome
    if (isPalindrome(head)) {
        printf("Linked list is a palindrome.\n");
    } else {
        printf("Linked list is not a palindrome.\n");
    }

    // Clean up the dynamically allocated memory (not shown in the
example)

    return 0;
}
```