7. Advanced Linked List Operations

7.1 Reverse a Singly Linked List

Method 1: Iterative

```
void reverseList(struct Node **head) {
    struct Node *prev = NULL;
    struct Node *current = *head;
    struct Node *next = NULL;

    while (current != NULL) {
        // Store next
        next = current->next;

        // Reverse current node's pointer
        current->next = prev;

        // Move pointers one position ahead
        prev = current;
        current = next;
    }

    *head = prev;
}
```

Visual Steps:

```
Initial: head → [10|●] → [20|●] → [30|NULL]

Step 1: prev=NULL, current=[10], next=[20]
NULL ← [10]   [20|●] → [30|NULL]

Step 2: prev=[10], current=[20], next=[30]
```

```
NULL ← [10] ← [20]   [30|NULL]

Step 3: prev=[20], current=[30], next=NULL
NULL ← [10] ← [20] ← [30]

Final: head → [30|●] → [20|●] → [10|NULL]
```

Time Complexity: O(n) **Space Complexity:** O(1)

Method 2: Recursive

```
struct Node* reverseRecursive(struct Node *head) {
    // Base case: empty list or single node
    if (head == NULL || head->next == NULL) {
        return head;
    }

    // Reverse the rest of the list
    struct Node *newHead = reverseRecursive(head->next);

    // Make next node point back
    head->next->next = head;
    head->next = NULL;

    return newHead;
}

// Wrapper function
void reverseListRecursive(struct Node **head) {
    *head = reverseRecursive(*head);
}
```

Time Complexity: O(n) **Space Complexity:** O(n) - recursion stack

7.2 Find Middle Element

Method 1: Two-Pass

```
int findMiddle(struct Node *head) {
    if (head == NULL) {
        printf("List is empty!\n");
```

```
        return -1;
    }

    // Count nodes
    int count = 0;
    struct Node *temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    // Go to middle
    temp = head;
    for (int i = 0; i < count / 2; i++) {
        temp = temp->next;
    }

    return temp->data;
}
```

Method 2: Slow-Fast Pointer (Optimal)

```
int findMiddleFast(struct Node *head) {
    if (head == NULL) {
        printf("List is empty!\n");
        return -1;
    }

    struct Node *slow = head;
    struct Node *fast = head;

    // Fast moves 2 steps, slow moves 1 step
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;
    }

    return slow->data;
}
```

Visual Steps:

```
List: [10] → [20] → [30] → [40] → [50] → NULL

Step 1: slow=[10], fast=[10]
Step 2: slow=[20], fast=[30]
Step 3: slow=[30], fast=[50]
Step 4: fast->next=NULL, stop

Middle: 30
```

Time Complexity: O(n) **Space Complexity:** O(1)

7.3 Detect Cycle (Floyd's Algorithm)

```
int hasCycle(struct Node *head) {
    if (head == NULL) {
        return 0;
    }

    struct Node *slow = head;
    struct Node *fast = head;

    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;

        // If they meet, there's a cycle
        if (slow == fast) {
            return 1;
        }
    }

    return 0;  // No cycle
}
```

Visual Representation:

```
Cycle Example:

      ┌──────────────┐
      ↓              │
[10] → [20] → [30] → [40]
```

```
                ↑     ↓
                └─────┘


Without cycle:
[10] → [20] → [30] → [40] → NULL
```

Time Complexity: O(n) **Space Complexity:** O(1)

7.4 Remove Duplicates from Sorted List

```
void removeDuplicates(struct Node *head) {
    if (head == NULL) {
        return;
    }

    struct Node *current = head;

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

Visual Steps:

Revision #1
Created 2025-10-27 05:05:41 UTC by DS
Updated 2025-10-27 05:06:36 UTC by DS