3. Singly Linked List Operations

3.1 Creating an Empty List

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

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

// Initialize head pointer
struct Node *head = NULL;
```

3.2 Insertion Operations

3.2.1 Insert at Beginning

```
void insertAtBeginning(struct Node **head, int value) {
    // Create new node
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));

    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }

    // Set data and next pointer
    newNode->data = value;
    newNode->next = *head;

    // Update head
    *head = newNode;
}
```

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

    insertAtBeginning(&head, 30);
    insertAtBeginning(&head, 20);
    insertAtBeginning(&head, 10);

    // List now: 10 → 20 → 30 → NULL

    return 0;
}
```

Visual Steps:

```
Initial: head → NULL

Step 1: Create node with data = 10
newNode → [10|NULL]

Step 2: Point newNode->next to current head
newNode → [10|●] → NULL

Step 3: Update head to newNode
head → [10|NULL]

Step 4: Insert 20
head → [20|●] → [10|NULL]

Step 5: Insert 30
head → [30|●] → [20|●] → [10|NULL]
```

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

3.2.2 Insert at End

```
void insertAtEnd(struct Node **head, int value) {
    // Create new node
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));

    if (newNode == NULL) {
```

```
        printf("Memory allocation failed!\n");
        return;
    }

    newNode->data = value;
    newNode->next = NULL;

    // If list is empty
    if (*head == NULL) {
        *head = newNode;
        return;
    }

    // Traverse to last node
    struct Node *temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }

    // Insert at end
    temp->next = newNode;
}
```

Visual Steps:

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

Step 1: Create newNode with data = 40
newNode → [40|NULL]

Step 2: Traverse to last node
temp moves: [10] → [20] → [30]

Step 3: Connect last node to newNode
head → [10|●] → [20|●] → [30|●] → [40|NULL]
```

Time Complexity: O(n) - must traverse entire list **Space Complexity:** O(1)

3.2.3 Insert at Position

```
void insertAtPosition(struct Node **head, int value, int position) {
    // Create new node
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));

    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }

    newNode->data = value;

    // Insert at beginning (position 0)
    if (position == 0) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

    // Traverse to position-1
    struct Node *temp = *head;
    for (int i = 0; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    // Check if position is valid
    if (temp == NULL) {
        printf("Invalid position!\n");
        free(newNode);
        return;
    }

    // Insert node
    newNode->next = temp->next;
    temp->next = newNode;
}
```

Visual Steps (Insert 25 at position 2):

```
Initial: head → [10|●] → [20|●] → [30|NULL]
                  0        1        2
```

```
Step 1: Create newNode [25|NULL]

Step 2: Traverse to position 1 (position - 1)
temp → [20|●]

Step 3: Connect newNode
newNode->next = temp->next  (points to 30)
[25|●] → [30|NULL]

Step 4: Connect temp to newNode
temp->next = newNode
head → [10|●] → [20|●] → [25|●] → [30|NULL]
```

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

3.3 Deletion Operations

3.3.1 Delete from Beginning

```
void deleteFromBeginning(struct Node **head) {
    // Check if list is empty
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    // Store current head
    struct Node *temp = *head;

    // Move head to next node
    *head = (*head)->next;

    // Free old head
    free(temp);
}
```

Visual Steps:

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

Step 1: temp = head
temp → [10|●] → [20|●] → [30|NULL]
head → [10|●] → [20|●] → [30|NULL]

Step 2: head = head->next
head → [20|●] → [30|NULL]
temp → [10|●] (to be freed)

Step 3: free(temp)
head → [20|●] → [30|NULL]
```

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

3.3.2 Delete from End

```
void deleteFromEnd(struct Node **head) {
    // Check if list is empty
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    // If only one node
    if ((*head)->next == NULL) {
        free(*head);
        *head = NULL;
        return;
    }

    // Traverse to second-last node
    struct Node *temp = *head;
    while (temp->next->next != NULL) {
        temp = temp->next;
    }

    // Delete last node
    free(temp->next);
    temp->next = NULL;
```

```
}
```

Visual Steps:

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

Step 1: Traverse to second-last node
temp → [20|●] → [30|NULL]

Step 2: Free last node
free(temp->next)

Step 3: Set second-last to NULL
head → [10|●] → [20|NULL]
```

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

3.3.3 Delete at Position

```
void deleteAtPosition(struct Node **head, int position) {
    // Check if list is empty
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    // Delete first node
    if (position == 0) {
        struct Node *temp = *head;
        *head = (*head)->next;
        free(temp);
        return;
    }

    // Traverse to position-1
    struct Node *temp = *head;
    for (int i = 0; i < position - 1 && temp != NULL; i++) {
        temp = temp->next;
    }

    // Check if position is valid
```

```
    if (temp == NULL || temp->next == NULL) {
        printf("Invalid position!\n");
        return;
    }

    // Delete node
    struct Node *nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    free(nodeToDelete);
}
```

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

3.3.4 Delete by Value

```
void deleteByValue(struct Node **head, int value) {
    // Check if list is empty
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }

    // If head node contains the value
    if ((*head)->data == value) {
        struct Node *temp = *head;
        *head = (*head)->next;
        free(temp);
        return;
    }

    // Search for the node
    struct Node *temp = *head;
    while (temp->next != NULL && temp->next->data != value) {
        temp = temp->next;
    }

    // If value not found
    if (temp->next == NULL) {
        printf("Value %d not found!\n", value);
        return;
    }
```

```
    // Delete node
    struct Node *nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    free(nodeToDelete);
}
```

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

3.4 Traversal and Display

```
void displayList(struct Node *head) {
    if (head == NULL) {
        printf("List is empty!\n");
        return;
    }

    struct Node *temp = head;
    printf("List: ");

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

// Alternative: Using for loop
void displayList2(struct Node *head) {
    printf("List: ");
    for (struct Node *temp = head; temp != NULL; temp = temp->next) {
        printf("%d → ", temp->data);
    }
    printf("NULL\n");
}
```

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

3.5 Searching

```
int search(struct Node *head, int value) {
    struct Node *temp = head;
    int position = 0;

    while (temp != NULL) {
        if (temp->data == value) {
            return position;  // Found at position
        }
        temp = temp->next;
        position++;
    }

    return -1;  // Not found
}

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

    insertAtEnd(&head, 10);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    int pos = search(head, 20);
    if (pos != -1) {
        printf("Found at position: %d\n", pos);
    } else {
        printf("Not found!\n");
    }

    return 0;
}
```

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

3.6 Length of List

```
int getLength(struct Node *head) {
    int count = 0;
    struct Node *temp = head;

    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    return count;
}

// Recursive version
int getLengthRecursive(struct Node *head) {
    if (head == NULL) {
        return 0;
    }
    return 1 + getLengthRecursive(head->next);
}
```

Time Complexity: O(n) **Space Complexity:** O(1) for iterative, O(n) for recursive

---

Revision #1
Created 2025-10-27 05:01:31 UTC by DS
Updated 2025-10-27 05:02:03 UTC by DS