Module 6 : Linked List

By the end of this module, students will be able to:

- Understand the concept and structure of linked lists

- Differentiate between arrays and linked lists

- Implement singly linked lists in C

- Perform basic operations: insertion, deletion, traversal, and searching

- Understand doubly linked lists and circular linked lists

- Apply linked lists to solve real-world problems

- Debug common linked list errors

- Analyze time and space complexity of linked list operations

- 1. Introduction to Linked Lists
- 2. Node Structure
- 3. Singly Linked List Operations
- 4. Complete Singly Linked List Example
- 5. Doubly Linked List
- 6. Circular Linked List
- 7. Advanced Linked List Operations
- 8. Practical Applications
- 9. Common Errors and Debugging

1. Introduction to Linked Lists

1.1 What is a Linked List?

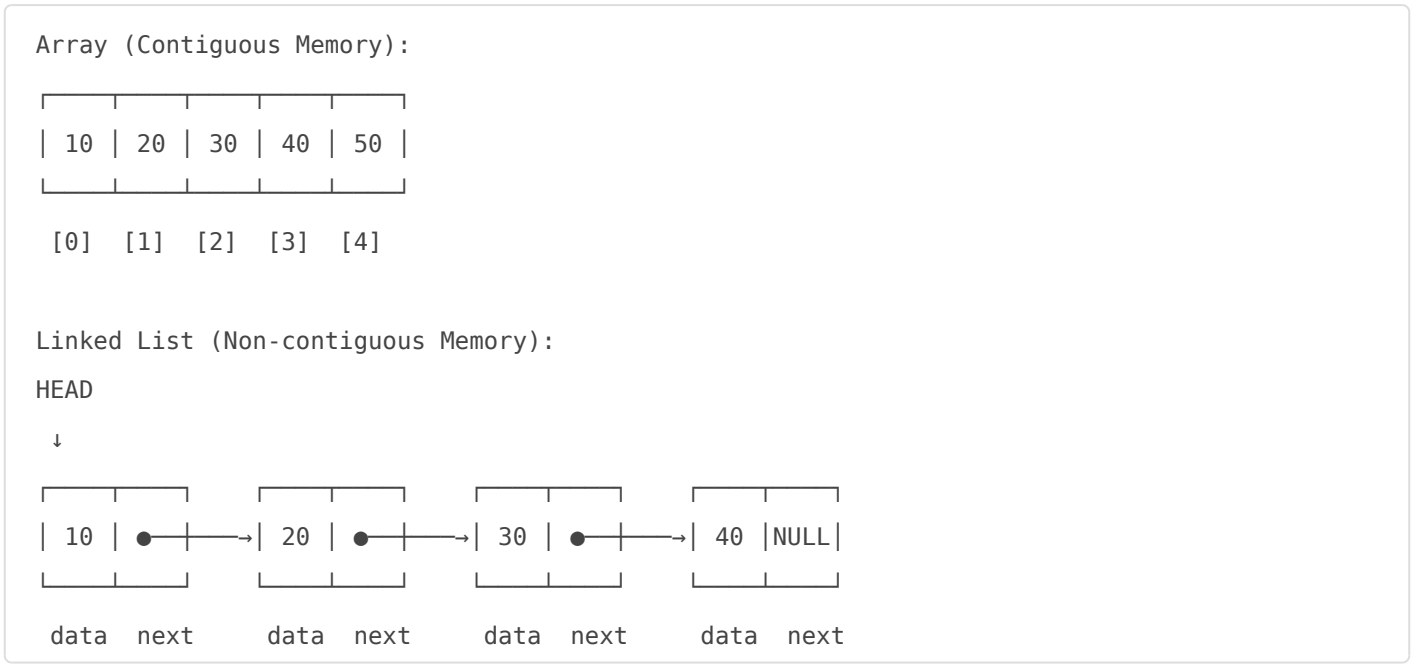
A **linked list** is a linear data structure where elements are not stored in contiguous memory locations. Instead, each element (called a **node**) contains:

1. **Data:** The actual value stored
2. **Pointer(s):** Reference to the next (and possibly previous) node

Analogy: Think of a linked list like a treasure hunt:

- Each clue (node) contains information (data)
- Each clue also tells you where to find the next clue (pointer)
- You start at the first clue (head)
- The last clue says "End of hunt" (NULL pointer)

Visual Representation:



1.2 Why Use Linked Lists?

Advantages:

1. **Dynamic Size:** Can grow or shrink at runtime
2. **Easy Insertion/Deletion:** No need to shift elements
3. **Memory Efficient:** Allocate memory only when needed

4. **Flexible Structure:** Can implement stacks, queues, graphs

Disadvantages:

1. **No Random Access:** Must traverse from beginning
2. **Extra Memory:** Requires space for pointers
3. **Sequential Access:** Slower than arrays for direct access
4. **Cache Performance:** Poor cache locality

1.3 Types of Linked Lists

```
1. Singly Linked List
   HEAD → [data|next] → [data|next] → [data|NULL]

2. Doubly Linked List
   HEAD ⇄ [prev|data|next] ⇄ [prev|data|next] ⇄ [prev|data|NULL]

3. Circular Linked List
   HEAD → [data|next] → [data|next] → [data|next] ──┐
    ↑                                               │
    └───────────────────────────────────────────────┘

4. Circular Doubly Linked List
   HEAD ⇄ [prev|data|next] ⇄ [prev|data|next] ──┐
    ↑                                           │
    └───────────────────────────────────────────┘
```

1.4 Linked List vs Array

Feature	Array	Linked List
Memory Allocation	Contiguous	Non-contiguous
Size	Fixed	Dynamic
Access Time	O(1)	O(n)
Insertion (beginning)	O(n)	O(1)
Insertion (end)	O(1)	O(n) or O(1) with tail
Deletion (beginning)	O(n)	O(1)
Memory Usage	Less (no pointers)	More (pointers)
Cache Performance	Better	Worse

Feature	Array	Linked List
Random Access	Yes	No

2. Node Structure

2.1 Defining a Node

In C, a node is typically defined using a **structure**:

```
// Definition of a node
struct Node {
    int data;           // Data part
    struct Node *next;  // Pointer to next node
};
```

Memory Layout:

```
Single Node in Memory:
┌─────────────┬─────────────┐
│   data      │   next      │
│  (int)      │ (pointer)   │
└─────────────┴─────────────┘
   4 bytes    8 bytes (on 64-bit system)
```

2.2 Creating a Node

Method 1: Static Allocation (Limited)

```
struct Node node1;
node1.data = 10;
node1.next = NULL;
```

Method 2: Dynamic Allocation (Recommended)

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

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

```
};

int main() {
    // Allocate memory for new node
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));

    // Check if allocation successful
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Initialize node
    newNode->data = 10;
    newNode->next = NULL;

    printf("Node created with data: %d\n", newNode->data);

    // Free memory when done
    free(newNode);

    return 0;
}
```

2.3 The Arrow Operator (->)

When working with pointers to structures, use the arrow operator:

```
struct Node *ptr;

// These are equivalent:
ptr->data = 10;      // Arrow operator (preferred)
(*ptr).data = 10;    // Dereference then dot operator
```

Why use ->?

- More readable
- Less typing
- Standard convention in C

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

4. Complete Singly Linked List Example

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

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

// Function prototypes
void insertAtBeginning(struct Node **head, int value);
void insertAtEnd(struct Node **head, int value);
void insertAtPosition(struct Node **head, int value, int position);
void deleteFromBeginning(struct Node **head);
void deleteFromEnd(struct Node **head);
void deleteAtPosition(struct Node **head, int position);
void deleteByValue(struct Node **head, int value);
void displayList(struct Node *head);
int search(struct Node *head, int value);
int getLength(struct Node *head);
void freeList(struct Node **head);

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

    printf("=== Linked List Operations Demo ===\n\n");

    // Insert operations
    printf("Inserting elements...\n");
    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 5);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 25);
```

```
    insertAtPosition(&head, 15, 2);

    printf("After insertions: ");
    displayList(head);
    printf("Length: %d\n\n", getLength(head));

    // Search operation
    int searchValue = 15;
    int pos = search(head, searchValue);
    if (pos != -1) {
        printf("Value %d found at position %d\n\n", searchValue, pos);
    } else {
        printf("Value %d not found\n\n", searchValue);
    }

    // Delete operations
    printf("Deleting from beginning...\n");
    deleteFromBeginning(&head);
    displayList(head);

    printf("Deleting from end...\n");
    deleteFromEnd(&head);
    displayList(head);

    printf("Deleting at position 1...\n");
    deleteAtPosition(&head, 1);
    displayList(head);

    printf("Deleting value 10...\n");
    deleteByValue(&head, 10);
    displayList(head);

    printf("\nFinal length: %d\n", getLength(head));

    // Clean up
    freeList(&head);
    printf("Memory freed.\n");

    return 0;
}
```

```
// Function implementations
void insertAtBeginning(struct Node **head, int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;
}

void insertAtEnd(struct Node **head, int value) {
    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 (*head == NULL) {
        *head = newNode;
        return;
    }

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

void insertAtPosition(struct Node **head, int value, int position) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }
```

```
    newNode->data = value;

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

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

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

    newNode->next = temp->next;
    temp->next = newNode;
}

void deleteFromBeginning(struct Node **head) {
    if (*head == NULL) {
        printf("List is empty!\n");
        return;
    }
    struct Node *temp = *head;
    *head = (*head)->next;
    free(temp);
}

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

    if ((*head)->next == NULL) {
```

```
        free(*head);
        *head = NULL;
        return;
    }

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

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

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

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

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

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

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

    if ((*head)->data == value) {
        struct Node *temp = *head;
        *head = (*head)->next;
        free(temp);
        return;
    }

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

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

    struct Node *nodeToDelete = temp->next;
    temp->next = nodeToDelete->next;
    free(nodeToDelete);
}

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

    struct Node *temp = head;
    while (temp != NULL) {
        printf("%d", temp->data);
        if (temp->next != NULL) {
            printf(" → ");
```

```
        }
        temp = temp->next;
    }
    printf(" → NULL\n");
}

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

    while (temp != NULL) {
        if (temp->data == value) {
            return position;
        }
        temp = temp->next;
        position++;
    }
    return -1;
}

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

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

void freeList(struct Node **head) {
    struct Node *temp;
    while (*head != NULL) {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}
```

5. Doubly Linked List

5.1 Node Structure

```
struct DNode {
    int data;
    struct DNode *prev;  // Pointer to previous node
    struct DNode *next;  // Pointer to next node
};
```

Visual Representation:

```
NULL ← [prev|10|next] ⇄ [prev|20|next] ⇄ [prev|30|next] → NULL
        ↑
       HEAD
```

5.2 Advantages of Doubly Linked List

1. **Bidirectional Traversal:** Can move forward and backward
2. **Easy Deletion:** Don't need previous node reference
3. **Easier Insertion:** Can insert before a node easily

Disadvantages:

1. **More Memory:** Extra pointer per node
2. **Complex Operations:** Must update two pointers

5.3 Basic Operations

5.3.1 Insert at Beginning

```
void insertAtBeginning(struct DNode **head, int value) {
    // Create new node
    struct DNode *newNode = (struct DNode*)malloc(sizeof(struct DNode));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }
```

```
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = *head;

    // Update head's prev if list not empty
    if (*head != NULL) {
        (*head)->prev = newNode;
    }

    *head = newNode;
}
```

Visual Steps:

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

Step 1: Create newNode [NULL|5|NULL]

Step 2: Connect newNode to head
newNode: [NULL|5|●] → [NULL|10|●]

Step 3: Update head's prev
[NULL|5|●] ⇄ [●|10|●]

Step 4: Update head
head → [NULL|5|●] ⇄ [●|10|●] ⇄ [●|20|NULL]
```

5.3.2 Insert at End

```
void insertAtEnd(struct DNode **head, int value) {
    struct DNode *newNode = (struct DNode*)malloc(sizeof(struct DNode));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }

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

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

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

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

5.3.3 Delete Node

```
void deleteNode(struct DNode **head, struct DNode *nodeToDelete) {
    // Check if list or node is NULL
    if (*head == NULL || nodeToDelete == NULL) {
        return;
    }

    // If node is head
    if (*head == nodeToDelete) {
        *head = nodeToDelete->next;
    }

    // Update next's prev pointer
    if (nodeToDelete->next != NULL) {
        nodeToDelete->next->prev = nodeToDelete->prev;
    }

    // Update prev's next pointer
    if (nodeToDelete->prev != NULL) {
        nodeToDelete->prev->next = nodeToDelete->next;
    }
```

```
    free(nodeToDelete);
}
```

5.3.4 Reverse Traversal

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

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

    // Traverse backward
    printf("List (reverse): ");
    while (temp != NULL) {
        printf("%d", temp->data);
        if (temp->prev != NULL) {
            printf(" ← ");
        }
        temp = temp->prev;
    }
    printf("\n");
}
```

6. Circular Linked List

6.1 Structure

In a circular linked list, the last node points back to the first node (or head).

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

Visual Representation:

6.2 Operations

6.2.1 Insert at Beginning

```
void insertAtBeginning(struct Node **head, int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    if (newNode == NULL) {
        printf("Memory allocation failed!\n");
        return;
    }

    newNode->data = value;

    // If list is empty
    if (*head == NULL) {
        newNode->next = newNode;  // Points to itself
        *head = newNode;
        return;
    }
```

```
    // Find last node
    struct Node *temp = *head;
    while (temp->next != *head) {
        temp = temp->next;
    }

    // Insert at beginning
    newNode->next = *head;
    temp->next = newNode;
    *head = newNode;
}
```

6.2.2 Display Circular List

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

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

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

    printf(" → (back to %d)\n", head->data);
}
```

6.2.3 Delete Node

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

```
}

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

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

    // Find last node
    while (temp->next != *head) {
        temp = temp->next;
    }

    // Delete head
    temp->next = (*head)->next;
    free(*head);
    *head = temp->next;
    return;
}

// Search for the node
prev = *head;
temp = (*head)->next;

while (temp != *head && temp->data != value) {
    prev = temp;
    temp = temp->next;
}

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

```
    // Delete node
    prev->next = temp->next;
    free(temp);
}
```

6.2.4 Count Nodes

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

    int count = 1;
    struct Node *temp = head->next;

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

    return count;
}
```

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:

8. Practical Applications

8.1 Polynomial Addition

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

// Node for polynomial term
struct PolyNode {
    int coeff;  // Coefficient
    int exp;    // Exponent
    struct PolyNode *next;
};

// Insert term in descending order of exponent
void insertTerm(struct PolyNode **poly, int coeff, int exp) {
    struct PolyNode *newNode = (struct PolyNode*)malloc(sizeof(struct PolyNode));
    newNode->coeff = coeff;
    newNode->exp = exp;
    newNode->next = NULL;

    if (*poly == NULL || (*poly)->exp < exp) {
        newNode->next = *poly;
        *poly = newNode;
    } else {
        struct PolyNode *temp = *poly;
        while (temp->next != NULL && temp->next->exp > exp) {
            temp = temp->next;
        }
        newNode->next = temp->next;
        temp->next = newNode;
    }
}

// Add two polynomials
struct PolyNode* addPolynomials(struct PolyNode *poly1, struct PolyNode *poly2) {
    struct PolyNode *result = NULL;
```

```
    while (poly1 != NULL && poly2 != NULL) {
        if (poly1->exp > poly2->exp) {
            insertTerm(&result, poly1->coeff, poly1->exp);
            poly1 = poly1->next;
        } else if (poly1->exp < poly2->exp) {
            insertTerm(&result, poly2->coeff, poly2->exp);
            poly2 = poly2->next;
        } else {
            // Same exponent - add coefficients
            int sumCoeff = poly1->coeff + poly2->coeff;
            if (sumCoeff != 0) {
                insertTerm(&result, sumCoeff, poly1->exp);
            }
            poly1 = poly1->next;
            poly2 = poly2->next;
        }
    }

    // Add remaining terms
    while (poly1 != NULL) {
        insertTerm(&result, poly1->coeff, poly1->exp);
        poly1 = poly1->next;
    }

    while (poly2 != NULL) {
        insertTerm(&result, poly2->coeff, poly2->exp);
        poly2 = poly2->next;
    }

    return result;
}

// Display polynomial
void displayPoly(struct PolyNode *poly) {
    if (poly == NULL) {
        printf("0\n");
        return;
    }
```

```
    while (poly != NULL) {
        printf("%dx^%d", poly->coeff, poly->exp);
        poly = poly->next;
        if (poly != NULL) {
            printf(" + ");
        }
    }
    printf("\n");
}

int main() {
    struct PolyNode *poly1 = NULL;
    struct PolyNode *poly2 = NULL;

    // Create first polynomial: 5x^3 + 4x^2 + 2
    insertTerm(&poly1, 5, 3);
    insertTerm(&poly1, 4, 2);
    insertTerm(&poly1, 2, 0);

    // Create second polynomial: 3x^3 + 2x + 1
    insertTerm(&poly2, 3, 3);
    insertTerm(&poly2, 2, 1);
    insertTerm(&poly2, 1, 0);

    printf("Polynomial 1: ");
    displayPoly(poly1);

    printf("Polynomial 2: ");
    displayPoly(poly2);

    struct PolyNode *result = addPolynomials(poly1, poly2);

    printf("Sum: ");
    displayPoly(result);

    return 0;
}

/* Output:
Polynomial 1: 5x^3 + 4x^2 + 2x^0
```

```
Polynomial 2: 3x^3 + 2x^1 + 1x^0
Sum: 8x^3 + 4x^2 + 2x^1 + 3x^0
*/
```

8.2 Music Playlist Implementation

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

struct Song {
    char title[100];
    char artist[100];
    int duration;  // in seconds
    struct Song *next;
};

// Add song to playlist
void addSong(struct Song **playlist, const char *title, const char *artist, int duration) {
    struct Song *newSong = (struct Song*)malloc(sizeof(struct Song));
    strcpy(newSong->title, title);
    strcpy(newSong->artist, artist);
    newSong->duration = duration;
    newSong->next = NULL;

    if (*playlist == NULL) {
        *playlist = newSong;
    } else {
        struct Song *temp = *playlist;
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newSong;
    }
}

// Display playlist
void displayPlaylist(struct Song *playlist) {
    if (playlist == NULL) {
```

```
        printf("Playlist is empty!\n");
        return;
    }

    int count = 1;
    printf("\n=== Playlist ===\n");
    while (playlist != NULL) {
        printf("%d. %s - %s (%d:%02d)\n",
               count++,
               playlist->title,
               playlist->artist,
               playlist->duration / 60,
               playlist->duration % 60);
        playlist = playlist->next;
    }
}

// Remove song by title
void removeSong(struct Song **playlist, const char *title) {
    if (*playlist == NULL) {
        printf("Playlist is empty!\n");
        return;
    }

    // If first song matches
    if (strcmp((*playlist)->title, title) == 0) {
        struct Song *temp = *playlist;
        *playlist = (*playlist)->next;
        free(temp);
        printf("Song removed: %s\n", title);
        return;
    }

    // Search for song
    struct Song *temp = *playlist;
    while (temp->next != NULL && strcmp(temp->next->title, title) != 0) {
        temp = temp->next;
    }

    if (temp->next == NULL) {
```

```
        printf("Song not found: %s\n", title);
        return;
    }

    struct Song *toRemove = temp->next;
    temp->next = toRemove->next;
    free(toRemove);
    printf("Song removed: %s\n", title);
}

// Calculate total duration
int totalDuration(struct Song *playlist) {
    int total = 0;
    while (playlist != NULL) {
        total += playlist->duration;
        playlist = playlist->next;
    }
    return total;
}

int main() {
    struct Song *myPlaylist = NULL;

    // Add songs
    addSong(&myPlaylist, "Bohemian Rhapsody", "Queen", 354);
    addSong(&myPlaylist, "Stairway to Heaven", "Led Zeppelin", 482);
    addSong(&myPlaylist, "Hotel California", "Eagles", 391);
    addSong(&myPlaylist, "Imagine", "John Lennon", 183);

    displayPlaylist(myPlaylist);

    int total = totalDuration(myPlaylist);
    printf("\nTotal duration: %d:%02d\n", total / 60, total % 60);

    // Remove a song
    removeSong(&myPlaylist, "Hotel California");

    displayPlaylist(myPlaylist);

    return 0;
```

```
}
```

8.3 Browser History (Back/Forward Navigation)

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

struct Page {
    char url[200];
    struct Page *prev;
    struct Page *next;
};

struct Browser {
    struct Page *current;
};

// Visit new page
void visitPage(struct Browser *browser, const char *url) {
    struct Page *newPage = (struct Page*)malloc(sizeof(struct Page));
    strcpy(newPage->url, url);
    newPage->next = NULL;

    if (browser->current != NULL) {
        // Clear forward history
        struct Page *temp = browser->current->next;
        while (temp != NULL) {
            struct Page *toDelete = temp;
            temp = temp->next;
            free(toDelete);
        }

        browser->current->next = newPage;
        newPage->prev = browser->current;
    } else {
        newPage->prev = NULL;
    }
```

```
    browser->current = newPage;
    printf("Visited: %s\n", url);
}

// Go back
void goBack(struct Browser *browser) {
    if (browser->current == NULL || browser->current->prev == NULL) {
        printf("Cannot go back!\n");
        return;
    }

    browser->current = browser->current->prev;
    printf("Back to: %s\n", browser->current->url);
}

// Go forward
void goForward(struct Browser *browser) {
    if (browser->current == NULL || browser->current->next == NULL) {
        printf("Cannot go forward!\n");
        return;
    }

    browser->current = browser->current->next;
    printf("Forward to: %s\n", browser->current->url);
}

// Display current page
void displayCurrent(struct Browser *browser) {
    if (browser->current == NULL) {
        printf("No page loaded!\n");
    } else {
        printf("Current page: %s\n", browser->current->url);
    }
}

int main() {
    struct Browser browser = {NULL};

    visitPage(&browser, "https://google.com");
    visitPage(&browser, "https://github.com");
```

```
    visitPage(&browser, "https://stackoverflow.com");

    printf("\n");
    goBack(&browser);
    goBack(&browser);

    printf("\n");
    goForward(&browser);

    printf("\n");
    visitPage(&browser, "https://reddit.com");

    printf("\n");
    displayCurrent(&browser);

    return 0;
}
```

9. Common Errors and Debugging

9.1 Memory Leaks

Problem:

```
void createList() {
    struct Node *head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 10;
    head->next = NULL;
    // MEMORY LEAK! head is lost when function returns
}
```

Solution:

```
struct Node* createList() {
    struct Node *head = (struct Node*)malloc(sizeof(struct Node));
    head->data = 10;
    head->next = NULL;
    return head;  // Return pointer to caller
}

// In main:
struct Node *myList = createList();
// ... use list ...
freeList(&myList);  // Free memory when done
```

9.2 Dereferencing NULL Pointer

Problem:

```
void insertAtEnd(struct Node **head, int value) {
    struct Node *temp = *head;
    while (temp->next != NULL) {  // CRASH if head is NULL!
        temp = temp->next;
```

```
    }
    // ...
}
```

Solution:

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

    if (*head == NULL) {  // Check for NULL first!
        *head = newNode;
        return;
    }

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

9.3 Lost Node References

Problem:

```
void deleteNode(struct Node **head, int position) {
    struct Node *temp = *head;
    for (int i = 0; i < position - 1; i++) {
        temp = temp->next;
    }
    temp->next = temp->next->next;  // MEMORY LEAK! Node not freed
}
```

Solution:

```
void deleteNode(struct Node **head, int position) {
    struct Node *temp = *head;
    for (int i = 0; i < position - 1; i++) {
```

```
        temp = temp->next;
    }
    struct Node *nodeToDelete = temp->next;  // Save reference
    temp->next = nodeToDelete->next;
    free(nodeToDelete);  // Free memory
}
```

9.4 Infinite Loop in Circular List

Problem:

```
void display(struct Node *head) {
    struct Node *temp = head;
    while (temp != NULL) {  // Never NULL in circular list!
        printf("%d ", temp->data);
        temp = temp->next;
    }
}
```

Solution:

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

    struct Node *temp = head;
    do {
        printf("%d ", temp->data);
        temp = temp->next;
    } while (temp != head);  // Check if back to head
}
```

9.5 Incorrect Pointer Updates

Problem:

```
void insertAtBeginning(struct Node *head, int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = head;
    head = newNode;  // Only changes local copy!
```

```
}
```

Solution:

```
void insertAtBeginning(struct Node **head, int value) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = *head;
    *head = newNode;  // Updates actual head pointer
}
```

9.6 Debugging Techniques

Print Node Addresses

```
void debugList(struct Node *head) {
    printf("\n=== Debug Info ===\n");
    struct Node *temp = head;
    int count = 0;

    while (temp != NULL) {
        printf("Node %d:\n", count++);
        printf("  Address: %p\n", (void*)temp);
        printf("  Data: %d\n", temp->data);
        printf("  Next: %p\n", (void*)temp->next);
        temp = temp->next;
    }
    printf("==================\n\n");
}
```

Check List Integrity

```
int checkListIntegrity(struct Node *head) {
    if (head == NULL) {
        return 1;  // Empty list is valid
    }

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

```
    // Check for cycles
    while (fast != NULL && fast->next != NULL) {
        slow = slow->next;
        fast = fast->next->next;

        if (slow == fast) {
            printf("ERROR: Cycle detected!\n");
            return 0;
        }
    }

    printf("List integrity: OK\n");
    return 1;
}
```

Visualize List

```
void visualizeList(struct Node *head) {
    if (head == NULL) {
        printf("NULL\n");
        return;
    }

    struct Node *temp = head;
    printf("HEAD → ");

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

9.7 Common Error Messages

Segmentation Fault:

- Dereferencing NULL pointer
- Accessing freed memory
- Buffer overflow

Memory Leak:

- Not freeing allocated memory
- Losing references to nodes
- Not calling free() on all nodes

Infinite Loop:

- Wrong termination condition
- Circular list without proper check
- Pointer not advancing

9.8 Preventive Measures

```
// Always check malloc return value
struct Node *newNode = (struct Node*)malloc(sizeof(struct Node));
if (newNode == NULL) {
    fprintf(stderr, "Memory allocation failed!\n");
    return NULL;
}

// Check for NULL before dereferencing
if (head != NULL) {
    // Safe to use head->data
}

// Free all nodes before exiting
void freeList(struct Node **head) {
    struct Node *temp;
    while (*head != NULL) {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}

// Set pointers to NULL after freeing
free(node);
```

```
node = NULL;

// Use assertions for debugging
#include <assert.h>
assert(head != NULL);  // Program stops if false
```