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");
}
```

Revision #1
Created 2025-10-27 05:04:00 UTC by DS
Updated 2025-10-27 05:04:26 UTC by DS