Singly Linked List in C

A **singly linked list** is a linear data structure where each element is a node, and each node points to the next node in the sequence.

Structure of a Node

Each node contains:

- `data` – the value of the node.
- `next` – a pointer to the next node.

Operations

a. Insert at Beginning

- Create a new node.
- Point new node's `next` to the current head.
- Update head to new node.

b. Insert at End

- Traverse to the last node.
- Point the last node's `next` to the new node.
- Set new node's `next` to `NULL`.

c. Insert After a Value

- Search for the node with the target value.
- Insert the new node after it.

d. Delete by Value

- If head contains the value, update head.
- Else, search and unlink the node.

e. Search

- Traverse and compare each node's data.

f. Display

- Print each node’s data from head to `NULL`.

Program in C

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

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

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// 1. Insert at the beginning
void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    newNode->next = *head;
    *head = newNode;
    printf("Inserted %d at the beginning.\n", value);

    /*
        Output:
        ---------------------------
        Inserted 10 at the beginning.
        Inserted 5 at the beginning.
    */
}

// 2. Insert at the end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
        printf("Inserted %d at the end (list was empty).\n", value);
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
    printf("Inserted %d at the end.\n", value);

    /*
        Output:
        ---------------------------
        Inserted 20 at the end.
        Inserted 30 at the end.
    */
```

```
}

// 3. Insert after a given value
void insertAfterValue(struct Node* head, int afterValue, int newValue) {
    struct Node* temp = head;
    while (temp != NULL && temp->data != afterValue) {
        temp = temp->next;
    }
    if (temp == NULL) {
        printf("Value %d not found in the list.\n", afterValue);
        return;
    }
    struct Node* newNode = createNode(newValue);
    newNode->next = temp->next;
    temp->next = newNode;
    printf("Inserted %d after %d.\n", newValue, afterValue);

    /*
        Output:
        ---------------------------
        Inserted 15 after 10.
    */
}

// 4. Delete a node by value
void deleteNode(struct Node** head, int value) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    // If head node itself holds the value
    if (temp != NULL && temp->data == value) {
        *head = temp->next;
        free(temp);
        printf("Deleted node with value %d (was head node).\n", value);

        /*
            Output:
            ---------------------------
            Deleted node with value 5 (was head node).
        */
        return;
    }

    // Search for the value to be deleted
    while (temp != NULL && temp->data != value) {
        prev = temp;
        temp = temp->next;
    }

    // If value was not found
    if (temp == NULL) {
        printf("Value %d not found in the list.\n", value);

        /*
```

```
                Output:
                --------------------------
                Value 100 not found in the list.
            */
            return;
        }

        // Unlink and delete the node
        prev->next = temp->next;
        free(temp);
        printf("Deleted node with value %d.\n", value);

        /*
            Output:
            --------------------------
            Deleted node with value 20.
        */
    }

    // 5. Search for an element
    void searchNode(struct Node* head, int value) {
        struct Node* temp = head;
        int position = 1;
        while (temp != NULL) {
            if (temp->data == value) {
                printf("Value %d found at position %d.\n", value, position);
                return;
            }
            temp = temp->next;
            position++;
        }
        printf("Value %d not found in the list.\n", value);

        /*
            Output:
            --------------------------
            Value 15 found at position 2.
            Value 100 not found in the list.
        */
    }

    // 6. Display the linked list
    void displayList(struct Node* head) {
        if (head == NULL) {
            printf("The list is empty.\n");
            return;
        }
        struct Node* temp = head;
        printf("Linked List: ");
        while (temp != NULL) {
            printf("%d -> ", temp->data);
            temp = temp->next;
        }
        printf("NULL\n");
```

```
        /*
            Example Output:
            ---------------------------
            Linked List: 5 -> 10 -> 20 -> 30 -> NULL
        */
    }

    // 7. Free all nodes (cleanup)
    void freeList(struct Node** head) {
        struct Node* temp;
        while (*head != NULL) {
            temp = *head;
            *head = (*head)->next;
            free(temp);
        }
    }

    // Main function to test all operations
    int main() {
        struct Node* head = NULL; // Initialize empty list

        // Demonstrating all operations
        insertAtBeginning(&head, 10);
        insertAtBeginning(&head, 5);
        insertAtEnd(&head, 20);
        insertAtEnd(&head, 30);
        displayList(head);

        insertAfterValue(head, 10, 15);
        displayList(head);

        deleteNode(&head, 5);
        displayList(head);

        deleteNode(&head, 20);
        displayList(head);

        searchNode(head, 15);
        searchNode(head, 100);

        freeList(&head); // Cleanup memory
        return 0;
    }



/*
    program output:
    -----------------------
    Inserted 10 at the beginning.
    Inserted 5 at the beginning.

    Inserted 20 at the end.
```

```
    Inserted 30 at the end.

    Linked List: 5 -> 10 -> 20 -> 30 -> NULL

    Inserted 15 after 10.
    Linked List: 5 -> 10 -> 15 -> 20 -> 30 -> NULL

    Deleted node with value 5 (was head node).
    Linked List: 10 -> 15 -> 20 -> 30 -> NULL

    Deleted node with value 20.
    Linked List: 10 -> 15 -> 30 -> NULL

    Value 15 found at position 2.
    Value 100 not found in the list.
*/
```

Doubly Linked List (DLL) in C

A **Doubly Linked List** is a linear data structure where each node contains links to both its previous and next nodes, allowing bidirectional traversal.

Structure of a Node

Each node contains:

- `data`: the value to store
- `prev`: pointer to the previous node
- `next`: pointer to the next node

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

Operations

a. Insert at Beginning

- Create a new node
- Set new node's `next` to current head
- Set current head's `prev` to new node (if head is not NULL)
- Update head to new node

b. Insert at End

- Traverse to the end of the list
- Set last node's `next` to new node
- Set new node's `prev` to the last node

c. Delete from Beginning

- Update head to `head->next`
- If head is not NULL, set `head->prev` to NULL
- Free the old head node

d. Delete from End

- Traverse to the last node
- Update second last node's `next` to NULL
- Free the last node

e. Delete by Value

- Search for the node with the given value
- Update `prev->next` and `next->prev` to unlink the node
- Free the target node

f. Search

- Traverse the list
- Compare each node's `data` with the target value

g. Display (Forward and Backward)

- **Forward Traversal**: From head to tail using `next` pointers
- **Backward Traversal**: From tail to head using `prev` pointers

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

// Define the structure for a node
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function to create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->prev = NULL;
    newNode->next = NULL;
```

```
    return newNode;
}

// 1. Insert at beginning
void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head != NULL) {
        newNode->next = *head;
        (*head)->prev = newNode;
    }
    *head = newNode;
    printf("Inserted %d at the beginning.\n", value);

    /*
        Output:
        ---------------------------
        Inserted 10 at the beginning.
        Inserted 5 at the beginning.
    */
}

// 2. Insert at end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        *head = newNode;
        printf("Inserted %d at the end (list was empty).\n", value);
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL)
        temp = temp->next;

    temp->next = newNode;
    newNode->prev = temp;
    printf("Inserted %d at the end.\n", value);

    /*
        Output:
        ---------------------------
        Inserted 20 at the end.
        Inserted 30 at the end.
    */
}

// 3. Delete from beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }
    struct Node* temp = *head;
    *head = (*head)->next;
    if (*head != NULL)
```

```
        (*head)->prev = NULL;

    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);

    /*
        Output:
        ----------------------------
        Deleted 5 from the beginning.
    */
}

// 4. Delete from end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

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

    if (temp->prev != NULL)
        temp->prev->next = NULL;
    else
        *head = NULL;

    printf("Deleted %d from the end.\n", temp->data);
    free(temp);

    /*
        Output:
        ----------------------------
        Deleted 30 from the end.
    */
}

// 5. Delete by value
void deleteByValue(struct Node** head, int value) {
    struct Node* temp = *head;

    // Traverse to find the node
    while (temp != NULL && temp->data != value)
        temp = temp->next;

    if (temp == NULL) {
        printf("Value %d not found in the list.\n", value);
        /*
            Output:
            ----------------------------
            Value 100 not found in the list.
        */
```

```
        return;
    }

    // If node to delete is head
    if (temp->prev == NULL)
        *head = temp->next;
    else
        temp->prev->next = temp->next;

    if (temp->next != NULL)
        temp->next->prev = temp->prev;

    printf("Deleted node with value %d.\n", temp->data);
    free(temp);

    /*
        Output:
        ---------------------------
        Deleted node with value 20.
    */
}

// 6. Search
void search(struct Node* head, int value) {
    int pos = 1;
    while (head != NULL) {
        if (head->data == value) {
            printf("Value %d found at position %d.\n", value, pos);
            return;
        }
        head = head->next;
        pos++;
    }
    printf("Value %d not found in the list.\n", value);

    /*
        Output:
        ---------------------------
        Value 10 found at position 1.
        Value 100 not found in the list.
    */
}

// 7. Display forward
void displayForward(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }
    printf("Forward: ");
    while (head != NULL) {
        printf("%d <-> ", head->data);
        if (head->next == NULL) break; // Save last for backward
        head = head->next;
```

```
    }
    printf("NULL\n");

    /*
        Output:
        ---------------------------
        Forward: 5 <-> 10 <-> 20 <-> 30 <-> NULL
    */
}

// 8. Display backward (from last node)
void displayBackward(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    // Go to last node
    while (head->next != NULL)
        head = head->next;

    printf("Backward: ");
    while (head != NULL) {
        printf("%d <-> ", head->data);
        head = head->prev;
    }
    printf("NULL\n");

    /*
        Output:
        ---------------------------
        Backward: 30 <-> 20 <-> 10 <-> 5 <-> NULL
    */
}

// 9. Free the list
void freeList(struct Node** head) {
    struct Node* temp;
    while (*head != NULL) {
        temp = *head;
        *head = (*head)->next;
        free(temp);
    }
}

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

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 5);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);
```

```
    displayForward(head);
    displayBackward(head);

    deleteFromBeginning(&head);
    displayForward(head);

    deleteFromEnd(&head);
    displayForward(head);

    deleteByValue(&head, 20);
    displayForward(head);

    deleteByValue(&head, 100); // not in list

    search(head, 10);
    search(head, 100);

    freeList(&head);
    return 0;
}

/*
    output:
    ---------------------------------
    Inserted 10 at the beginning.
    Inserted 5 at the beginning.

    Inserted 20 at the end.
    Inserted 30 at the end.

    Forward: 5 <-> 10 <-> 20 <-> 30 <-> NULL
    Backward: 30 <-> 20 <-> 10 <-> 5 <-> NULL

    Deleted 5 from the beginning.
    Forward: 10 <-> 20 <-> 30 <-> NULL

    Deleted 30 from the end.
    Forward: 10 <-> 20 <-> NULL

    Deleted node with value 20.
    Forward: 10 <-> NULL

    Value 100 not found in the list.

    Value 10 found at position 1.

    Value 100 not found in the list.
*/
```

Circular Linked List (CLL) in C

A **Circular Linked List** is a variation of a singly linked list in which the last node points back to the first node, forming a circle. This structure allows continuous traversal from any node.

Structure of a Node

Each node contains:

- `data`: to store the value
- `next`: pointer to the next node (last node points back to head)

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

Operations in Circular Linked List (CLL)

a. Insert at Beginning

- Create a new node
- Set new node's `next` to current head
- Traverse to last node and set last node's `next` to new node
- Update head to new node

b. Insert at End

- Create a new node
- Traverse to last node
- Set last node's `next` to new node
- Set new node's `next` to head

c. Delete from Beginning

- Find the last node
- Point last node's `next` to `head->next`
- Free the old head
- Update head to `head->next`

d. Delete from End

- Find second last node
- Set its `next` to head
- Free the last node

e. Delete by Value

- Special handling for head, tail, and middle nodes

- Traverse the list and unlink the target node by updating `next` pointers
- Free the deleted node

f. Search

- Traverse using a loop until returning to head
- Compare each node's `data` with the target value

g. Display

- Use a `do-while` loop to traverse and print nodes
- Stop when the traversal comes back to head (completes the full circle)

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

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

// Create a new node
struct Node* createNode(int value) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    newNode->next = NULL;
    return newNode;
}

// 1. Insert at beginning
void insertAtBeginning(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
    } else {
        struct Node* temp = *head;
        while (temp->next != *head)
            temp = temp->next;

        newNode->next = *head;
        temp->next = newNode;
        *head = newNode;
    }
    printf("Inserted %d at the beginning.\n", value);

    /*
        Output:
        ----------------------------
        Inserted 10 at the beginning.
```

```
        Inserted 5 at the beginning.
    */
}

// 2. Insert at end
void insertAtEnd(struct Node** head, int value) {
    struct Node* newNode = createNode(value);
    if (*head == NULL) {
        newNode->next = newNode;
        *head = newNode;
        printf("Inserted %d at the end (list was empty).\n", value);
        return;
    }
    struct Node* temp = *head;
    while (temp->next != *head)
        temp = temp->next;

    temp->next = newNode;
    newNode->next = *head;
    printf("Inserted %d at the end.\n", value);

    /*
        Output:
        ------------------------------
        Inserted 20 at the end.
        Inserted 30 at the end.
    */
}

// 3. Delete from beginning
void deleteFromBeginning(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node* temp = *head;
    if (temp->next == temp) {
        printf("Deleted %d (only node).\n", temp->data);
        free(temp);
        *head = NULL;
        return;
    }

    struct Node* last = *head;
    while (last->next != *head)
        last = last->next;

    last->next = temp->next;
    *head = temp->next;
    printf("Deleted %d from the beginning.\n", temp->data);
    free(temp);

    /*
```

```
        Output:
        ------------------------------
        Deleted 5 from the beginning.
    */
}

// 4. Delete from end
void deleteFromEnd(struct Node** head) {
    if (*head == NULL) {
        printf("List is empty. Nothing to delete.\n");
        return;
    }

    struct Node* temp = *head;

    if (temp->next == temp) {
        printf("Deleted %d (only node).\n", temp->data);
        free(temp);
        *head = NULL;
        return;
    }

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

    prev->next = *head;
    printf("Deleted %d from the end.\n", temp->data);
    free(temp);

    /*
        Output:
        ------------------------------
        Deleted 30 from the end.
    */
}

// 5. Delete by value
void deleteByValue(struct Node** head, int value) {
    if (*head == NULL) {
        printf("List is empty.\n");
        return;
    }

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

    // Only node
    if (temp->data == value && temp->next == *head) {
        free(temp);
        *head = NULL;
        printf("Deleted node with value %d (only node).\n", value);
        return;
```

```
    }

    // Head node
    if (temp->data == value) {
        while (temp->next != *head)
            temp = temp->next;
        struct Node* del = *head;
        temp->next = (*head)->next;
        *head = (*head)->next;
        free(del);
        printf("Deleted node with value %d (was head node).\n", value);
        return;
    }

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

    if (temp == *head) {
        printf("Value %d not found in the list.\n", value);
        /*
            Output:
            ------------------------------
            Value 100 not found in the list.
        */
        return;
    }

    prev->next = temp->next;
    free(temp);
    printf("Deleted node with value %d.\n", value);

    /*
        Output:
        ------------------------------
        Deleted node with value 20.
    */
}

// 6. Search
void search(struct Node* head, int value) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    int pos = 1;
    do {
        if (temp->data == value) {
            printf("Value %d found at position %d.\n", value, pos);
```

```
            return;
        }
        temp = temp->next;
        pos++;
    } while (temp != head);

    printf("Value %d not found in the list.\n", value);

    /*
        Output:
        ------------------------------
        Value 15 found at position 2.
        Value 100 not found in the list.
    */
}

// 7. Display the list
void display(struct Node* head) {
    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    struct Node* temp = head;
    printf("Circular List: ");
    do {
        printf("%d -> ", temp->data);
        temp = temp->next;
    } while (temp != head);
    printf("(head)\n");

    /*
        Output:
        ------------------------------
        Circular List: 5 -> 10 -> 20 -> 30 -> (head)
    */
}

// 8. Free all nodes
void freeList(struct Node** head) {
    if (*head == NULL) return;

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

    *head = NULL;
}

// Main Function
```

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

    insertAtBeginning(&head, 10);
    insertAtBeginning(&head, 5);
    insertAtEnd(&head, 20);
    insertAtEnd(&head, 30);

    display(head);

    deleteFromBeginning(&head);
    display(head);

    deleteFromEnd(&head);
    display(head);

    deleteByValue(&head, 20);
    display(head);

    deleteByValue(&head, 100); // Not in list

    search(head, 10);
    search(head, 100);

    freeList(&head);
    return 0;
}


/*
    Output:
    -----------------------------------
    Inserted 10 at the beginning.
    Inserted 5 at the beginning.

    Inserted 20 at the end.
    Inserted 30 at the end.

    Circular List: 5 -> 10 -> 20 -> 30 -> (head)

    Deleted 5 from the beginning.
    Circular List: 10 -> 20 -> 30 -> (head)

    Deleted 30 from the end.
    Circular List: 10 -> 20 -> (head)

    Deleted node with value 20.
    Circular List: 10 -> (head)

    Value 100 not found in the list.

    Value 10 found at position 1.
    Value 100 not found in the list.
```

```
 */
```