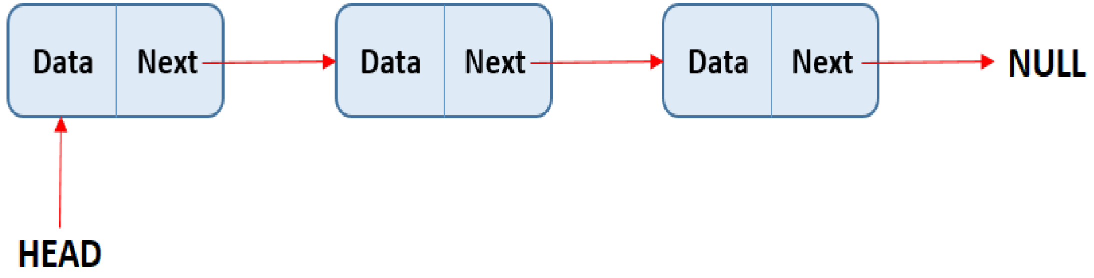
Linked Lists

Introduction

- A linked list is a linear data structure where each element (node) contains a data part and a reference (or link) to the next node in the sequence.

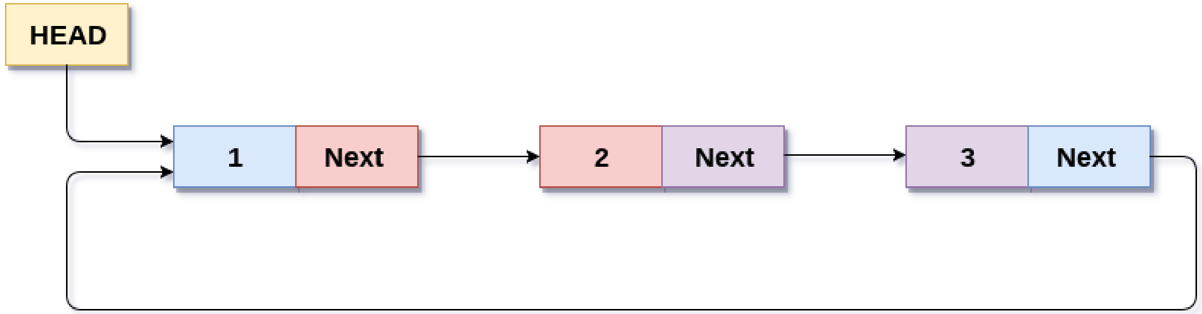
Types of Linked Lists

- Singly Linked List:
  - Each node points to the next node.
  - Last node points to **NULL**



- Doubly Linked List:
  - Each node points to both the next and previous nodes.
  - Allows traversal in both directions.

- Circular Linked List:
  - Last node points back to the first node, forming a circle.
  - Can be singly or doubly linked.



Node Structure

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

Components:

- **Pointer/Link:** The reference to the next node in the list.
- **Data:** The value stored in the node.

Basic Operations on Linked Lists

1. Creation
   - Initializing an empty list.
   - Allocating nodes dynamically.
2. Insertion
   - At the beginning, end, or after a specific node
   - Adjusting pointers accordingly.
3. Deletion
   - Removing nodes from the beginning, end, or a specific position.
   - Managing memory deallocation.
4. Traversal

   Visiting each node in the list to process data.

Advanced Operations

1. Reversal
2. Concatenation
3. Searching
4. Sorting

Advantages of Linked Lists

- **Dynamic Size:** Linked lists can grow and shrink in size dynamically.
- **Efficient Insertions/Deletions:** Unlike arrays, linked lists can easily insert or delete nodes without shifting elements.
- **No Wasted Memory:** Linked lists allocate memory as needed, avoiding wasted space.

Disadvantages of Linked Lists

- **Memory Overhead:** Extra memory is required for storing pointers.
- **Slower Access:** Random access is not possible; traversal is required to access an element.
- **Complexity:** More complex to implement than arrays due to pointer management.

Applications of Linked Lists

- **Dynamic Memory Allocation:** Managing memory in operating systems.
- **Implementing Stacks and Queues:** Used in data structures like stacks, queues, and graphs.
- **Polynomial Arithmetic:** Used to store coefficients of polynomials for efficient operations.

Real-World Example

- Music Playlist
- Image Viewer
- Web Browser History
- Undo/Redo Functionality in Text Editors
- Implementation of Queues in Printers

Linked list code

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

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

// Function to create a new node
struct Node* createNode(int data) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    if (!newNode) {
        printf("Memory error\n");
        return NULL;
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

```c
// Function to insert a node at the beginning of the linked list
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    newNode->next = *head;
    *head = newNode;
}

// Function to insert a node at the end of the linked list
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = createNode(data);
    if (*head == NULL) {
        *head = newNode;
        return;
    }
    struct Node* temp = *head;
    while (temp->next != NULL) {
        temp = temp->next;
    }
    temp->next = newNode;
}
```

```c
// Function to insert a node at a given position in the linked list
void insertAtPosition(struct Node** head, int data, int position) {
    struct Node* newNode = createNode(data);

    // If inserting at the beginning
    if (position == 1) {
        newNode->next = *head;
        *head = newNode;
        return;
    }

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

    // If the position is beyond the length of the list
    if (temp == NULL) {
        printf("Position out of bounds\n");
        free(newNode);
        return;
    }

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

```c
// Function to display the linked list
void printList(struct Node* head) {
    struct Node* temp = head;
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

```c
// Function to delete a node with a given key
void deleteNode(struct Node** head, int key) {
    struct Node* temp = *head;
    struct Node* prev = NULL;

    // If head node itself holds the key to be deleted
    if (temp != NULL && temp->data == key) {
        *head = temp->next; // Changed head
        free(temp);         // Free old head
        return;
    }

    // Search for the key to be deleted, keep track of the previous
        node
    while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
    }

    // If key was not present in the linked list
    if (temp == NULL) return;

    // Unlink the node from linked list
    prev->next = temp->next;
    free(temp);
}
```

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

    insertAtEnd(&head, 1);
    insertAtEnd(&head, 2);
    insertAtEnd(&head, 3);
    insertAtBeginning(&head, 0);
    insertAtPosition(&head, 4, 3);  // Inserting 4 at position 3
    printList(head);

    deleteNode(&head, 2);
    printList(head);

    return 0;
}
```

Menu Driven Program of Singly Linked List

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

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

// Function prototypes
void insertAtBeginning();
void insertAtEnd();
void deleteNode();
void displayList();
void searchList();

struct Node* head = NULL; // Initial head of the list (empty)
```

```c
// Function to insert a node at the beginning of the list
void insertAtBeginning() {
    int value;
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));

    printf("Enter the value to insert: ");
    scanf("%d", &value);
    newNode->data = value;
    newNode->next = head;
    head = newNode;
    printf("Node inserted at the beginning.\n");
}
```

```c
void insertAtEnd() {
    int value;
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* temp = head;

    printf("Enter the value to insert: ");
    scanf("%d", &value);
    newNode->data = value;
    newNode->next = NULL;

    if (head == NULL) {
        head = newNode;
    } else {
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newNode;
    }
    printf("Node inserted at the end.\n");
}
```

```c
// Function to delete a node with a specific value
void deleteNode() {
    int value;
    struct Node* temp = head;
    struct Node* prev = NULL;

    printf("Enter the value to delete: ");
    scanf("%d", &value);

    if (temp != NULL && temp->data == value) {
        head = temp->next;  // Head node to be deleted
        free(temp);
        printf("Node with value %d deleted.\n", value);
        return;
    }

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

    if (temp == NULL) {
        printf("Node with value %d not found.\n", value);
        return;
    }

    prev->next = temp->next;
    free(temp);
    printf("Node with value %d deleted.\n", value);
}
```

```c
// Function to display the entire list
void displayList() {
    struct Node* temp = head;

    if (head == NULL) {
        printf("List is empty.\n");
        return;
    }

    printf("Linked List: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
```

```c
// Function to search for a specific value in the list
void searchList() {
    int value, position = 1;
    struct Node* temp = head;

    printf("Enter the value to search: ");
    scanf("%d", &value);

    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);
}
```

```c
// Main function to implement menu-driven program
int main() {
    int choice;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Delete a Node\n");
        printf("4. Display List\n");
        printf("5. Search List\n");
        printf("6. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                insertAtBeginning();
                break;
            case 2:
                insertAtEnd();
                break;
            case 3:
                deleteNode();
                break;
            case 4:
                displayList();
                break;
            case 5:
                searchList();
                break;
            case 6:
                printf("Exiting program.\n");
                exit(0);
            default:
                printf("Invalid choice. Please try again.\n");
        }
    }

    return 0;
}
```

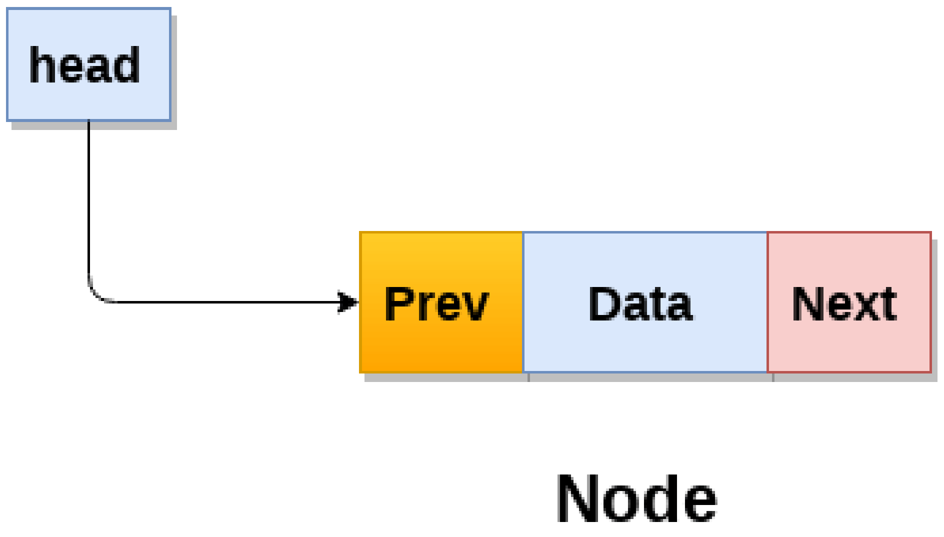
What is a Doubly Linked List?

- **Definition**

  A linked list where each node points to both the previous and the next node.
- **Structure of a Node:**
  - **Data**: Stores value.
  - **Previous Pointer**: Points to the previous node.
  - **Next Pointer**: Points to the next node.

Node Structure in C

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



data: Stores the element.
prev: Points to the previous node.
next: Points to the next node.

Advantages of Doubly Linked List

- Allows both forward and backward traversal.
- Easier deletion of a node compared to singly linked list.
- More flexible for complex data structures (e.g., deque, navigation systems).

Disadvantages

- Requires more memory due to extra pointer.
- Slightly more complex than a singly linked list.

Insertion at Beginning

```
void insertAtBeginning(Node** head, int newData) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = newData;
    newNode->prev = NULL;
    newNode->next = *head;

    if (*head != NULL)
        (*head)->prev = newNode;

    *head = newNode;
}
```

Insertion at End

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

    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }

    while (last->next != NULL)
        last = last->next;

    last->next = newNode;
    newNode->prev = last;
}
```

Deletion from Beginning

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

    Node* temp = *head;
    *head = (*head)->next;

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

    free(temp);
}
```

Traversal of Doubly Linked List

```
void traverse(Node* head) {
    Node* last;
    printf("Forward Traversal: ");
    while (head != NULL) {
        printf("%d ", head->data);
        last = head;
        head = head->next;
    }

    printf("\nBackward Traversal: ");
    while (last != NULL) {
        printf("%d ", last->data);
        last = last->prev;
    }
}
```

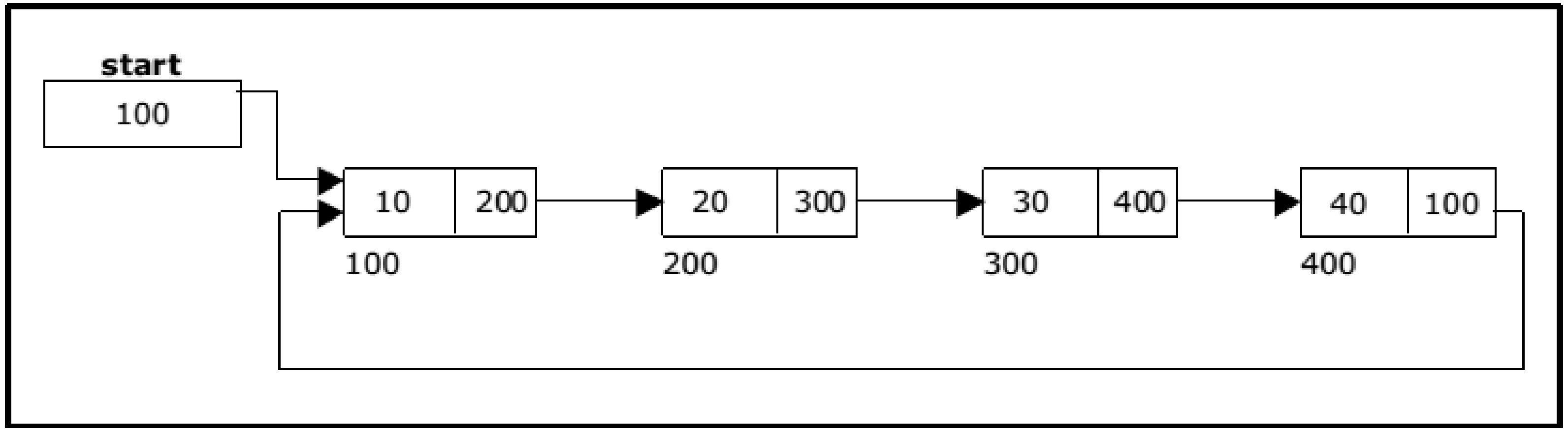
What is a Circular Linked List?

- **Definition**

  A circular linked list is a linked list where the last node points to the first node, forming a circle.

- **Structure of a Node:**
  - **Data**: Stores value.
  - **Next Pointer**: Points to the next node.

Node Structure in C



Insertion at the beginning

```c
// Insert a new node at the beginning
void insertAtBeginning(struct Node** head, int data) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    struct Node* temp = *head;

    newNode->data = data; // Assign data to the new node

    if (*head == NULL) {  // If the list is empty
        *head = new_node;
        newNode->next = newNode; // Link the node to itself
    } else {
        // Find the last node
        while (temp->next != *head) {
            temp = temp->next;
        }
        newNode->next = *head;  // Point the new node to the old head
        temp->next = newNode;  // Point the last node to the new node
        *head = new_node;         // Update head to new node
    }
}
```

Insert at the end

```c
// Insert a new node at the end
void insertAtEnd(struct Node** head, int data) {
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    struct Node* temp = *head;

    newNode->data = data;
    newNode->next = *head; // The new node will point to the head

    if (*head == NULL) {  // If the list is empty
        *head = newNode;
        newNode->next = newNode;  // Point the new node to itself
    } else {
        // Find the last node
        while (temp->next != *head) {
            temp = temp->next;
        }
        temp->next = newNode;  // Last node points to the new node
    }
}
```

Traversing of List

```c
// Function to print the circular linked list
void display(struct Node* head) {
    struct Node* temp = head;

    if (head != NULL) {
        do {
            printf("%d -> ", temp->data);
            temp = temp->next;
        } while (temp != head);
        printf("(back to head)\n");
    } else {
        printf("List is empty\n");
    }
}
```

Difference between singly and circular linked list

Feature	Singly Linked List	Circular Linked List
End Connection	Last node points to NULL.	Last node points to the first node.
Traversal	Ends when NULL is encountered.	Continues indefinitely in a loop.
Memory Usage	Requires a NULL pointer at the end.	No NULL, but needs extra management for circularity.
Applications	Used in linear structures (stacks, queues).	Used in circular processes (e.g., round-robin scheduling).
Complexity of Operations	Simple, straightforward.	Requires careful handling to maintain the loop.
Insertions/Deletions	Easier to implement at ends.	Requires adjustments to maintain circular links.