Linked Lists in C

A linked list is a dynamic data structure that consists of a sequence of nodes, where each node contains data and a pointer to the next node in the sequence. This structure allows for flexible memory allocation and efficient insertion and deletion operations.

Basic Operations

1. **Initialization:**
    - Create a new linked list by initializing a head pointer to `NULL`.
2. **Insertion:**
    - **At the beginning:** Create a new node, set its next pointer to the current head, and update the head pointer.
    - **At the end:** Traverse the list to find the last node, create a new node, set its next pointer to `NULL`, and set the last node's next pointer to the new node.
    - **After a specific node:** Traverse the list to find the node before the insertion point, create a new node, set its next pointer to the node after the insertion point, and set the previous node's next pointer to the new node.
3. **Deletion:**

- **At the beginning:** Update the head pointer to the next node and free the deleted node.
- **At the end:** Traverse the list to find the second-to-last node, set its next pointer to `NULL`, and free the deleted node.
- **After a specific node:** Traverse the list to find the node before the deletion point, set its next pointer to the node after the deletion point, and free the deleted node.

4. **Traversal:**
   - Start from the head node and iterate through the list until the current node's next pointer is `NULL`.
5. **Searching:**
   - Traverse the list and compare each node's data with the target value. If a match is found, return the node's address; otherwise, return `NULL`.

Code Example

C

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

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

struct Node *head = NULL;
```

```c
void insertAtBeginning(int data) {
    struct Node *newNode = (struct Node*)malloc(sizeof(struct
Node));
    newNode->data = data;
    newNode->next = head;
    head = newNode;
}

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

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

// ... other operations (deletion, traversal, searching) ...

int main() {
    // ... insert elements, perform operations ...
    return 0;
}
```

Key Points

- Linked lists are dynamic data structures that can grow or shrink as needed.
- Operations like insertion, deletion, traversal, and searching can be implemented efficiently.

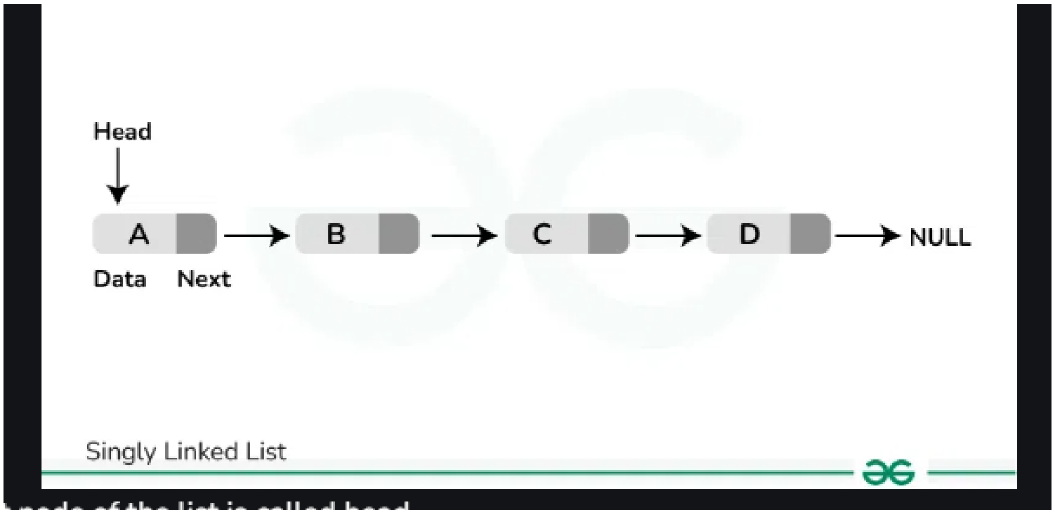
- Memory management is crucial for linked lists to avoid memory leaks.
- Linked lists are often used in various applications, such as implementing stacks, queues, and graphs.

Types of Linked Lists

Linked lists can be classified based on their structure and the order in which elements are accessed:

1. Singly Linked List:

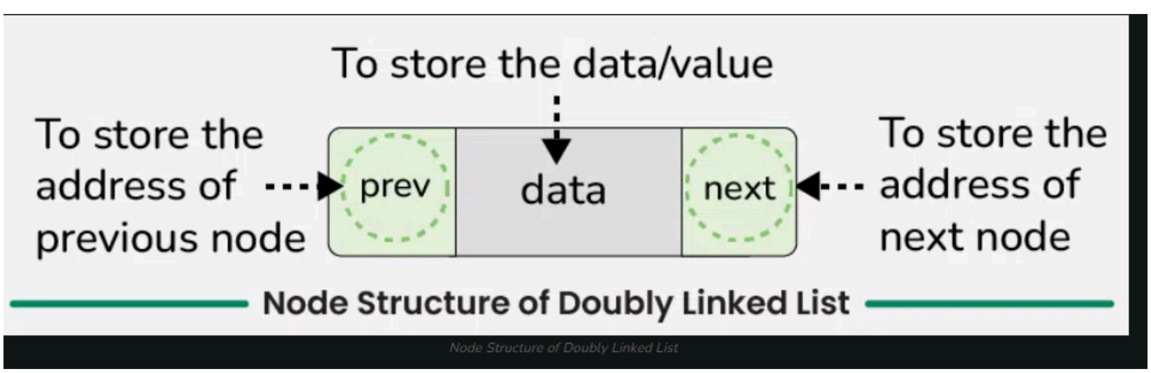
- **Structure:** Each node has a data field and a pointer to the next node.
- **Access:** Elements can be accessed sequentially from the head to the tail.
- **Operations:** Insertion, deletion, and traversal can be performed efficiently at the beginning or end of the list.
- **Image:**
- **Next pointer:** Points to the next node in the list.



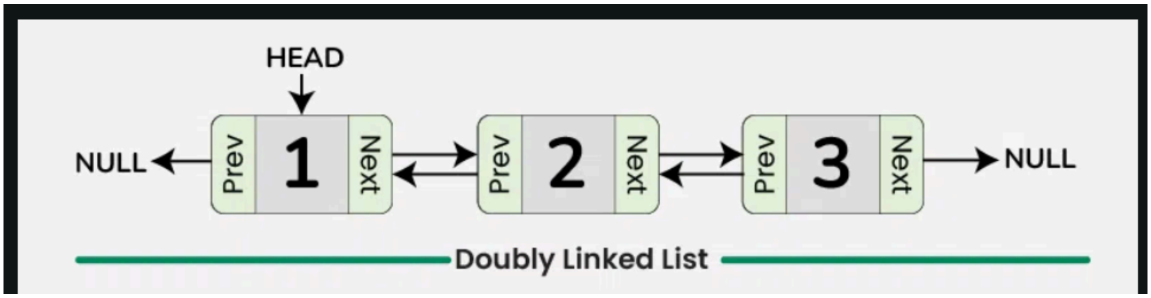
Operations:

2. Doubly Linked List:

- **Structure:** Each node has a data field, a pointer to the previous node, and a pointer to the next node.
- **Access:** Elements can be accessed sequentially in both directions.
- **Operations:** Insertion, deletion, and traversal can be performed efficiently at any position in the list.
- **Image:**



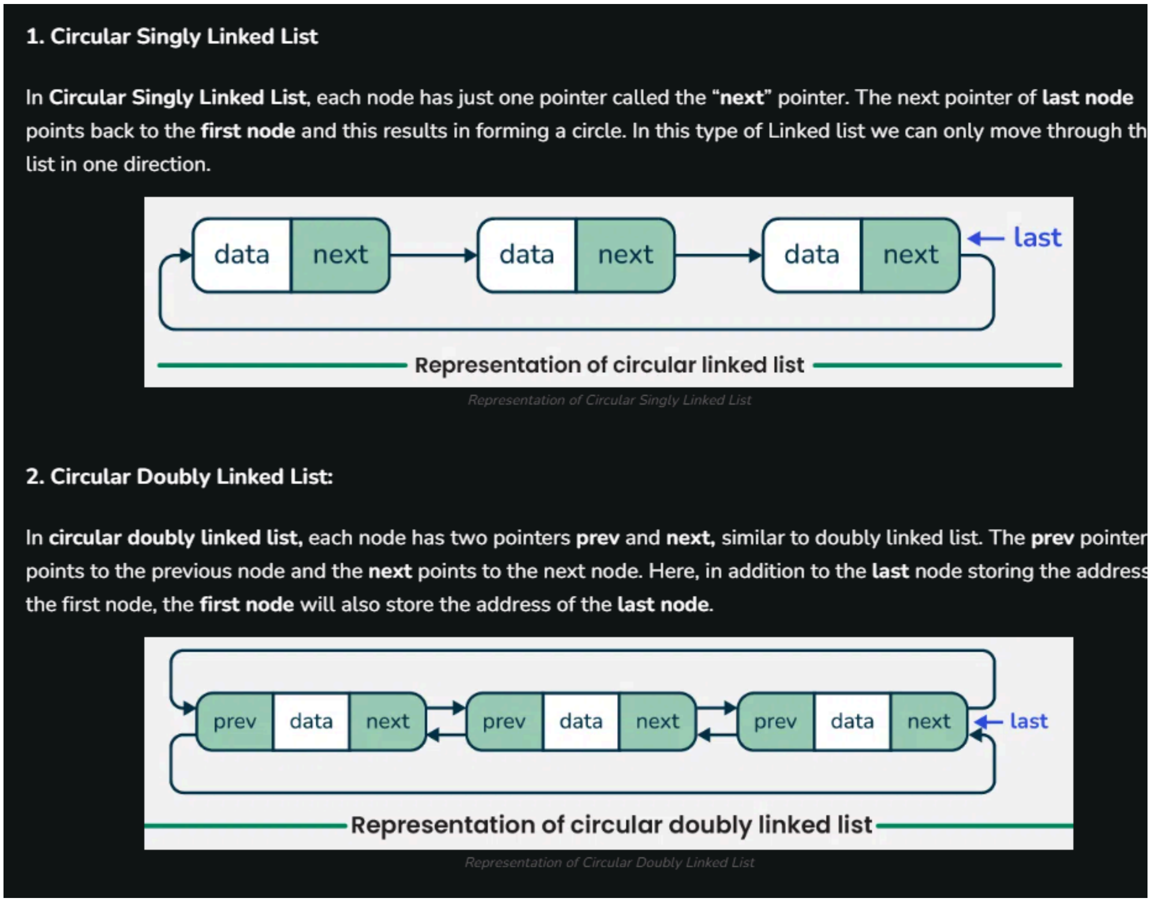
Node Structure of Doubly Linked List

- 

3. Circular Linked List:

- **Structure:** The last node's next pointer points back to the first node, forming a circular structure.
- **Access:** Elements can be accessed sequentially in both directions, but there is no defined beginning or end.
- **Operations:** Insertion and deletion can be performed efficiently at any position in the list.

- **Image:**



1. **Circular Singly Linked List**

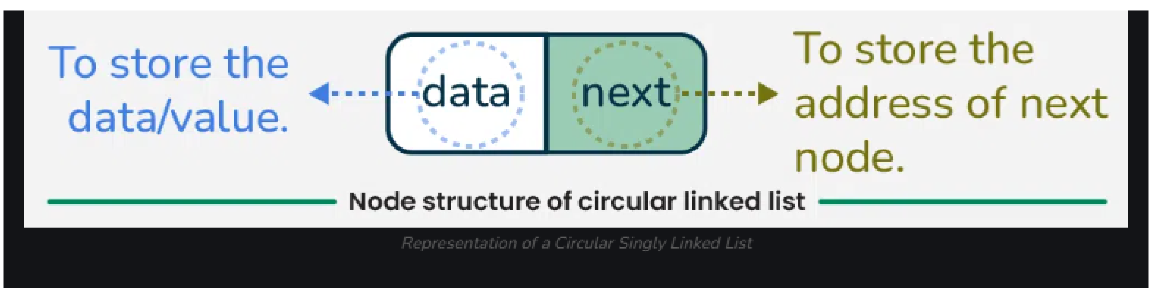
In **Circular Singly Linked List**, each node has just one pointer called the "**next**" pointer. The next pointer of **last node** points back to the **first node** and this results in forming a circle. In this type of Linked list we can only move through th list in one direction.

Representation of Circular Singly Linked List

2. **Circular Doubly Linked List:**

In **circular doubly linked list,** each node has two pointers **prev** and **next,** similar to doubly linked list. The **prev** pointer points to the previous node and the **next** points to the next node. Here, in addition to the **last** node storing the address the first node, the **first node** will also store the address of the **last node**.

Representation of Circular Doubly Linked List

- 

Representation of a Circular Singly Linked List

-

Choosing the right type of linked list depends on the specific requirements of your application. Consider factors such as:

- **Access patterns:** Whether you need to access elements in both directions or only sequentially.
- **Insertion and deletion operations:** How frequently you need to insert or delete elements, and at what positions.
- **Memory usage:** Whether you need to minimize memory usage or can afford additional pointers.