Understanding Linked Lists

A **linked list** is a linear data structure where elements (called nodes) are connected using pointers. Unlike arrays, linked lists do not store elements in contiguous memory locations. Each node contains two parts:

1. **Data**: The actual value of the node.
2. **Pointer**: A reference (or pointer) to the next node.

Types of Linked Lists

1. **Singly Linked List (SLL)**: Each node points to the next node. The last node points to `nullptr`.
2. **Doubly Linked List (DLL)**: Each node points to both the previous and next nodes.
3. **Circular Linked List (CLL)**: The last node points back to the first node, forming a circle.

---

1. Singly Linked List

Structure of a Singly Linked List Node

- Each node contains:
    i. `data`: The value stored in the node.
    ii. `next`: A pointer to the next node.

Code for a Singly Linked List

```cpp
#include <iostream>
using namespace std;

// Node definition
class Node {
public:
    int data;       // The data part of the node
    Node* next;     // Pointer to the next node

    // Constructor to initialize a node
    Node(int val) {
        data = val;         // Set the data field of the node to the provided value
        next = nullptr;     // Initialize the next pointer to null
    }
};
```

```cpp
// Singly Linked List Class
class SinglyLinkedList {
private:
    Node* head;     // Pointer to the first node of the list

public:
    SinglyLinkedList() {
        head = nullptr;  // Initialize the list as empty (head points to null)
    }

    // Add a node at the end of the list
    void append(int val) {
        Node* newNode = new Node(val);  // Create a new node with the given value
        if (!head) {                    // If the list is empty (head is null)
            head = newNode;             // Set the new node as the head of the list
            return;                     // Exit the function
        }
        Node* temp = head;              // Start from the head of the list
        while (temp->next) {            // Traverse the list until the last node
            temp = temp->next;          // Move to the next node
        }
        temp->next = newNode;           // Link the last node to the new node
    }

    // Traverse and print all elements
    void traverse() {
        Node* temp = head;              // Start from the head of the list
        while (temp) {                  // While there are more nodes to traverse
            cout << temp->data << " -> "; // Print the data in the current node
            temp = temp->next;          // Move to the next node
        }
        cout << "NULL" << endl;         // Print NULL to indicate the end of the list
    }
};

int main() {
    SinglyLinkedList list;

    list.append(10);
    list.append(20);
    list.append(30);

    cout << "Singly Linked List: ";
    list.traverse();

    return 0;
}
```

Key Points

- **Node**: Represents an element in the list.

- **Head**: The first node of the list. It is a pointer that starts the chain.
- **Traversing**: Start from `head` and follow the `next` pointers until you reach `nullptr`.
- **Appending**: To add a node, traverse to the last node and link the new node.

---

2. Doubly Linked List

Structure of a Doubly Linked List Node

- Each node contains:
  i. `data`: The value stored in the node.
  ii. `next`: Pointer to the next node.
  iii. `prev`: Pointer to the previous node.

Code for a Doubly Linked List

```
#include <iostream>
using namespace std;

// Node definition
class Node {
public:
    int data;       // The data part of the node
    Node* next;     // Pointer to the next node
    Node* prev;     // Pointer to the previous node

    // Constructor to initialize a node
    Node(int val) {
        data = val;         // Set the data field of the node to the provided value
        next = nullptr;     // Initialize the next pointer to null
        prev = nullptr;     // Initialize the previous pointer to null
    }
};

// Doubly Linked List Class
class DoublyLinkedList {
private:
    Node* head;     // Pointer to the first node of the list

public:
    DoublyLinkedList() {
        head = nullptr;  // Initialize the list as empty (head points to null)
    }

    // Add a node at the end of the list
    void append(int val) {
        Node* newNode = new Node(val);  // Create a new node with the given value
```

```cpp
        if (!head) {                        // If the list is empty (head is null)
            head = newNode;                 // Set the new node as the head of the list
            return;                         // Exit the function
        }
        Node* temp = head;                  // Start from the head of the list
        while (temp->next) {                // Traverse the list until the last node
            temp = temp->next;              // Move to the next node
        }
        temp->next = newNode;               // Link the last node to the new node
        newNode->prev = temp;               // Link the new node back to the last node
    }

    // Traverse and print all elements (forward)
    void traverseForward() {
        Node* temp = head;                  // Start from the head of the list
        while (temp) {                      // While there are more nodes to traverse
            cout << temp->data << " <-> "; // Print the data in the current node
            temp = temp->next;              // Move to the next node
        }
        cout << "NULL" << endl;             // Print NULL to indicate the end of the list
    }

    // Traverse and print all elements (backward)
    void traverseBackward() {
        if (!head) return;                  // If the list is empty, exit the function
        Node* temp = head;                  // Start from the head of the list

        // Move to the last node
        while (temp->next) {                // Traverse the list until the last node
            temp = temp->next;              // Move to the next node
        }

        // Traverse backward
        while (temp) {                      // While there are more nodes to traverse backward
            cout << temp->data << " <-> "; // Print the data in the current node
            temp = temp->prev;              // Move to the previous node
        }
        cout << "NULL" << endl;             // Print NULL to indicate the start of the list
    }
};

int main() {
    DoublyLinkedList list;

    list.append(10);
    list.append(20);
    list.append(30);

    cout << "Doubly Linked List (Forward): ";
    list.traverseForward();

    cout << "Doubly Linked List (Backward): ";
    list.traverseBackward();
```

```cpp
    return 0;
}
```

Key Points

- **Node**: Stores the value and has pointers to both previous and next nodes.
- **Traversing Forward**: Start from `head` and follow the `next` pointers.
- **Traversing Backward**: Start from the last node and follow the `prev` pointers.

---

3. Circular Linked List

Structure of a Circular Linked List Node

- Each node contains:
  i. `data`: The value stored in the node.
  ii. `next`: Pointer to the next node. The last node points back to the first node.

Code for a Circular Linked List

```cpp
#include <iostream>
using namespace std;

// Node definition
class Node {
public:
    int data;       // The data part of the node
    Node* next;     // Pointer to the next node

    // Constructor to initialize a node
    Node(int val) {
        data = val;         // Set the data field of the node to the provided value
        next = nullptr;     // Initialize the next pointer to null
    }
};

// Circular Linked List Class
class CircularLinkedList {
private:
    Node* head;     // Pointer to the first node of the list

public:
    CircularLinkedList() {
        head = nullptr;  // Initialize the list as empty (head points to null)
    }
```

```cpp
    // Add a node at the end of the list
    void append(int val) {
        Node* newNode = new Node(val);  // Create a new node with the given value
        if (!head) {                    // If the list is empty (head is null)
            head = newNode;             // Set the new node as the head of the list
            newNode->next = head;       // Make it circular by pointing the new node to itself
            return;                     // Exit the function
        }
        Node* temp = head;              // Start from the head of the list
        while (temp->next != head) {    // Traverse the list until the last node (next points
            temp = temp->next;          // Move to the next node
        }
        temp->next = newNode;           // Link the last node to the new node
        newNode->next = head;           // Point the new node back to the head, making the lis
    }

    // Traverse and print all elements
    void traverse() {
        if (!head) return;              // If the list is empty, exit the function
        Node* temp = head;              // Start from the head of the list
        do {                            // Use a do-while loop to handle circular traversal
            cout << temp->data << " -> "; // Print the data in the current node
            temp = temp->next;          // Move to the next node
        } while (temp != head);         // Stop when the traversal reaches the head again
        cout << "(head)" << endl;       // Indicate that the traversal ends at the head
    }
};

int main() {
    CircularLinkedList list;

    list.append(10);
    list.append(20);
    list.append(30);

    cout << "Circular Linked List: ";
    list.traverse();

    return 0;
}
```

Key Points

- **Node**: Each node contains a `data` value and a pointer (`next`).
- **Head**: The first node. It is pointed to by the last node.
- **Traversing**: Use a `do-while` loop to ensure the `head` is visited.

Comparison of Linked Lists

Feature	Singly Linked List	Doubly Linked List	Circular Linked List
Direction	Forward only	Forward and Backward	Forward or Backward (depending on implementation)
Memory Usage	Less (1 pointer)	More (2 pointers)	Moderate (1 pointer)
Traversing End	Ends at `nullptr`	Ends at `nullptr`	Ends when it loops back to `head`

Syntax Comparison Between Singly, Doubly, and Circular Linked Lists

Here's a detailed comparison of the syntax and structure of **Singly Linked List (SLL)**, **Doubly Linked List (DLL)**, and **Circular Linked List (CLL)**:

---

1. Node Definition

Feature	Singly Linked List (SLL)	Doubly Linked List (DLL)	Circular Linked List (CLL)
Data	`data`	`data`	`data`
Pointer(s)	One pointer (`Node* next`)	Two pointers (`Node* next` , `Node* prev`)	One pointer (`Node* next`)

Code Example:

```
// Singly Linked List Node
class SinglyNode {
public:
    int data;          // Stores the data value of the node
    SinglyNode* next; // Pointer to the next node in the list

    // Constructor to initialize the node with a value
    SinglyNode(int val) {
        data = val;       // Assign the value to the node's data field
        next = nullptr;   // Initialize the next pointer to null
    }
};
```

```cpp
// Doubly Linked List Node
class DoublyNode {
public:
    int data;          // Stores the data value of the node
    DoublyNode* next;  // Pointer to the next node in the list
    DoublyNode* prev;  // Pointer to the previous node in the list

    // Constructor to initialize the node with a value
    DoublyNode(int val) {
        data = val;         // Assign the value to the node's data field
        next = nullptr;     // Initialize the next pointer to null
        prev = nullptr;     // Initialize the previous pointer to null
    }
};

// Circular Linked List Node
class CircularNode {
public:
    int data;          // Stores the data value of the node
    CircularNode* next; // Pointer to the next node in the list (loops back to head)

    // Constructor to initialize the node with a value
    CircularNode(int val) {
        data = val;        // Assign the value to the node's data field
        next = nullptr;    // Initialize the next pointer to null (to be set later to head)
    }
};
```

---

2. Head Pointer

Feature	SLL	DLL	CLL
Head Pointer	Points to the first node	Points to the first node	Points to the first node
Tail Pointer	Optional (used for optimized append)	Optional (used for optimized append)	Points to the head to form a circle

Code Example:

```cpp
Node* head = nullptr;  // Common to all
Node* tail = nullptr;  // Optional in SLL/DLL but useful for CLL
```

---

3. Appending a Node

Step	SLL	DLL	CLL
Create Node	`Node* newNode = new Node(val);`	`Node* newNode = new Node(val);`	`Node* newNode = new Node(val);`
Traverse to End	Traverse using `next` until `nullptr`.	Traverse using `next` until `nullptr`.	Traverse using `next` until `next == head`.
Update Pointers	`last->next = newNode;`	`last->next = newNode;` `newNode->prev = last;`	`last->next = newNode;` `newNode->next = head;`

Code Example:

```
// Singly Linked List
void appendSLL(Node*& head, int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next) {
        temp = temp->next;
    }
    temp->next = newNode;
}

// Doubly Linked List
void appendDLL(Node*& head, int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}

// Circular Linked List
void appendCLL(Node*& head, int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = newNode;
        newNode->next = head; // Point to itself
        return;
    }
    Node* temp = head;
```

```
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
```

---

4. Traversing the List

Feature	SLL	DLL	CLL
Start	From `head`	From `head`	From `head`
Condition	Until `temp != nullptr`	Until `temp != nullptr`	Until `temp->next != head` (loop back)
Direction	Forward only	Forward and backward	Forward only

Code Example:

```
// Singly Linked List Traversal
void traverseSLL(Node* head) {
    Node* temp = head;
    while (temp) {
        cout << temp->data << " -> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

// Doubly Linked List Traversal (Forward)
void traverseDLL(Node* head) {
    Node* temp = head;
    while (temp) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

// Circular Linked List Traversal
void traverseCLL(Node* head) {
    if (!head) return;
    Node* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
```

```
    cout << "(head)" << endl;
}
```

---

5. Deleting a Node

Feature	SLL	DLL	CLL
Pointer Update	Update `next` of the previous node to skip the target.	Update both `prev` and `next` pointers.	Update the last node's `next` to skip the target.

Code Example (Deletion by Value):

```
// Singly Linked List Deletion
void deleteSLL(Node*& head, int key) {
    if (!head) return;
    if (head->data == key) {  // Deleting head
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }
    Node* temp = head;
    while (temp->next && temp->next->data != key) {
        temp = temp->next;
    }
    if (temp->next) {
        Node* toDelete = temp->next;
        temp->next = temp->next->next;
        delete toDelete;
    }
}

// Doubly Linked List Deletion
void deleteDLL(Node*& head, int key) {
    if (!head) return;
    if (head->data == key) {  // Deleting head
        Node* temp = head;
        head = head->next;
        if (head) head->prev = nullptr;
        delete temp;
        return;
    }
    Node* temp = head;
    while (temp && temp->data != key) {
        temp = temp->next;
    }
    if (temp) {
```

```
        if (temp->next) temp->next->prev = temp->prev;
        if (temp->prev) temp->prev->next = temp->next;
        delete temp;
    }
}

// Circular Linked List Deletion
void deleteCLL(Node*& head, int key) {
    if (!head) return;
    if (head->data == key) {  // Deleting head
        Node* temp = head;
        Node* tail = head;
        while (tail->next != head) {
            tail = tail->next;
        }
        if (head == tail) {  // Only one node
            head = nullptr;
        } else {
            tail->next = head->next;
            head = head->next;
        }
        delete temp;
        return;
    }
    Node* temp = head;
    while (temp->next != head && temp->next->data != key) {
        temp = temp->next;
    }
    if (temp->next != head) {
        Node* toDelete = temp->next;
        temp->next = temp->next->next;
        delete toDelete;
    }
}
```

Summary Table of Syntax

Operation	SLL Syntax	DLL Syntax	CLL Syntax
Node Definition	`Node* next;`	`Node* next; Node* prev;`	`Node* next;`
Appending	`temp->next = newNode;`	`temp->next = newNode; newNode->prev = temp;`	`temp->next = newNode; newNode->next = head;`
Traversal Condition	`while (temp != nullptr)`	`while (temp != nullptr)`	`while (temp->next != head)`