More types of Linked Lists

By: Aaryan Vinod Kumar

What are Doubly Linked Lists?

A doubly linked list is a type of linked list where each node contains two pointers — one pointing to the next node and one pointing to the previous node. This allows traversal in both forward and backward directions.

Structure of a Node

```
struct Node {
    int data;      // Value stored in the node
    Node* next;    // Pointer to the next node
    Node* prev;    // Pointer to the previous node
};
```

How it Works

Each node is linked to both its next and previous nodes. The first node's `prev` is `NULL`, and the last node's `next` is `NULL`.

Basic Operations on Doubly Linked List

1. Traversing Forward and Backward

```
void printForward(Node* head) {
    Node* temp = head;
    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->next;
    }
    cout << "NULL" << endl;
}

void printBackward(Node* tail) {
    Node* temp = tail;
    while (temp != nullptr) {
        cout << temp->data << " <-> ";
        temp = temp->prev;
    }
    cout << "NULL" << endl;
}
```

2. Inserting at Head

```
void insertAtHead(Node*& head, int value) {
    Node* newNode = new Node{value, head, nullptr};
    if (head != nullptr) head->prev = newNode;
    head = newNode;
}
```

3. Inserting at Tail

```
void insertAtTail(Node*& head, int value) {
    Node* newNode = new Node{value, nullptr, nullptr};
    if (head == nullptr) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next != nullptr) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->prev = temp;
}
```

4. Deleting a Node

```
void deleteNode(Node*& head, int value) {
    Node* temp = head;
    while (temp != nullptr && temp->data != value) {
        temp = temp->next;
    }
    if (temp == nullptr) return;

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

    if (temp->next != nullptr) temp->next->prev =
temp->prev;
```

```
    delete temp;
}
```

What are Circular Linked Lists?

A circular linked list is a list where the last node points back to the first node instead of `NULL`.

Types

- Singly Circular: Last node points to the head.
- Doubly Circular: Both first and last nodes link to each other.

Structure of a Node (Singly Circular)

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

In a singly circular list: `tail->next = head;`

Operations in Circular Linked List

1. Traversal

```
void printList(Node* head) {
    if (head == nullptr) return;
    Node* temp = head;
    do {
        cout << temp->data << " -> ";
        temp = temp->next;
    } while (temp != head);
    cout << "(back to head)" << endl;
}
```

2. Insertion at Tail

```
void insertAtTail(Node*& head, int value) {
    Node* newNode = new Node{value, nullptr};
    if (head == nullptr) {
        head = newNode;
        newNode->next = head;
        return;
    }
    Node* temp = head;
    while (temp->next != head) {
        temp = temp->next;
    }
    temp->next = newNode;
    newNode->next = head;
}
```

What are Header Linked Lists?

A header linked list has a special dummy node called the header, which does not store actual data but simplifies edge case handling.

Structure

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

struct HeaderList {
    Node* head; // This points to the dummy node
};
```

- First node is a dummy node.
- Actual data starts from `head->next`.

Note: The head of a header linked list can also be used to store the information regarding the list itself, though it is not recommended

Advantages of Header Nodes

- Avoids null checks at the beginning of the list.
- Simplifies insertion and deletion logic.
- Makes the list uniform and predictable.

Sample Use Case (Header List)

```
void insert(HeaderList& list, int value) {
    Node* newNode = new Node{value, list.head->next};
    list.head->next = newNode;
}
```

Real-World Use Cases

- Doubly Linked List: Browser history, undo/redo stacks.
- Circular List: CPU task scheduling, round-robin games.
- Header List: Academic implementation for simplifying logic.

Advantages

- Doubly Linked List: Two-way traversal, flexible deletion.
- Circular List: No NULL termination — always active.
- Header List: Cleaner, safer insertions/deletions.

Disadvantages

- More pointers = more memory use.
- Care needed to handle circular links (infinite loops).
- Slightly more complex insertion logic.

Best Practices

- Always free memory using `delete`.
- Carefully update both `next` and `prev` for DLLs.
- Use a dummy node to reduce edge case errors in Header Lists.