# Singly Linked Lists

By: Aaryan Vinod Kumar

## What are Singly Linked Lists?

A **linked list** is a linear data structure where each element (called a **node**) points to the next element in the sequence. Unlike arrays, the elements are **not** stored in contiguous memory locations.

A **singly linked list** is a type of linked list where each node points **only to the next node**.

### Structure of a Node

To define a node in C++, we use a `struct`:

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

Pointer: A pointer is a variable that stores the memory address of another variable. In linked lists, we use pointers to link nodes together.

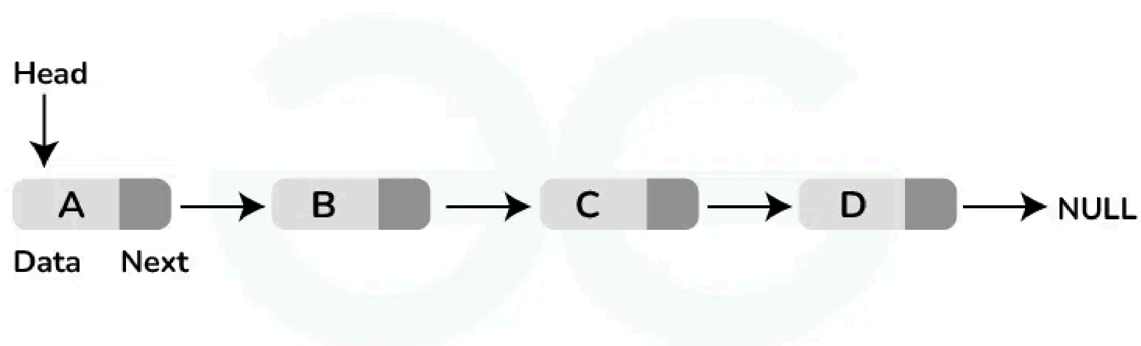
## What is a `struct`?

A `struct` in C++ is a **user-defined data type** that lets you group variables of different types. Here, `struct Node` groups:

- `data`: the value of the node.
- `next`: a pointer to another node of the same type.

## How a Singly Linked List Works

Each node knows **only about the next node**. The list ends when a node's `next` is `NULL`.



## Basic Operations on Singly Linked List:

1. Traversing the List (Printing Elements)

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

How it works:

- We use a temporary pointer (`temp`) to move through the list.
- In each loop, we print the current node's data and move to the next node.
- The loop stops when we hit `nullptr` (i.e., the end of the list).

2. Inserting at the Head (Beginning)

```
void insertAtHead(Node*& head, int value) {
    Node* newNode = new Node{value, head};
    head = newNode;
}
```

**Explanation:**

- We create a new node with the given value.
- Set its `next` pointer to the current head.
- Update the `head` pointer to this new node.

This ensures the new node becomes the first in the list.

3. Inserting at the Tail (End)

```
void insertAtTail(Node*& head, int value) {
    Node* newNode = new Node{value, nullptr};

    if (head == nullptr) {
        head = newNode;
        return;
    }

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

Explanation:

- If the list is empty, the new node becomes the head.
- Otherwise, we move through the list to the last node (`temp->next == nullptr`).
- We set the `next` of the last node to point to the new node.

4. Deleting a Node by Value

```
void deleteNode(Node*& head, int value) {
    if (head == nullptr) return;

    if (head->data == value) {
        Node* temp = head;
        head = head->next;
        delete temp;
        return;
    }

    Node* curr = head;
    while (curr->next != nullptr && curr->next->data !=
value) {
        curr = curr->next;
    }

    if (curr->next == nullptr) return;

    Node* temp = curr->next;
    curr->next = curr->next->next;
```

```
    delete temp;
}
```

**Explanation:**

- First, check if the list is empty.
- If the value is in the head, delete it and move the head.
- Otherwise, traverse to the node before the one to be deleted.
- Update its `next` to skip the deleted node.
- Use `delete` to free memory and avoid memory leaks.

## Sample Program: Creating and Using a Singly Linked List

```cpp
#include <iostream>

using namespace std;


struct Node {

    int data;

    Node* next;

};


void printList(Node* head) {

    Node* temp = head;

    while (temp != nullptr) {

        cout << temp->data << " -> ";

        temp = temp->next;

    }

    cout << "NULL" << endl;

}


void insertAtHead(Node*& head, int value) {
```

```
    Node* newNode = new Node{value, head};
    head = newNode;
}

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

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

```cpp
        Node* temp = head;

        head = head->next;

        delete temp;

        return;

    }

    Node* curr = head;

    while (curr->next != nullptr && curr->next->data != value) {

        curr = curr->next;

    }

    if (curr->next == nullptr) return;

    Node* temp = curr->next;

    curr->next = curr->next->next;

    delete temp;

}


int main() {

    Node* head = nullptr;


    insertAtTail(head, 10);

    insertAtTail(head, 20);
```

```
    insertAtTail(head, 30);


    cout << "Original List:\n";

    printList(head);


    insertAtHead(head, 5);

    cout << "After Inserting 5 at Head:\n";

    printList(head);


    deleteNode(head, 20);

    cout << "After Deleting 20:\n";

    printList(head);


    return 0;

}
```

## Real-World Use Cases

- **Web Browsers**: Back and forward navigation.
- **Music Apps**: Queue of songs.
- **Undo Feature**: Text editors like Word use it to store changes.

## Advantages

- Dynamically sized — no memory wastage.
- Fast insertions/deletions at the beginning or middle.

## Disadvantages

- No direct/random access (O(n) traversal).
- Uses extra memory (pointers for each node).

## Best Practices Section

- Always check if `head == nullptr` before operations.
- Always free memory using `delete` when removing nodes to avoid memory leaks.

## Practice Problem

Q: Build a list using `insertAtTail()`. Then insert a value at the head and another at the tail. Print the final list.