1. Singly Linked Lists

Explanation

A **Singly Linked List** is a linear data structure where elements (called *nodes*) are connected using pointers.
Each node contains:

- **Data** (the value)
- **Pointer (next)** → points to the next node

Unlike arrays:

- Memory is **not continuous**
- Size can grow or shrink dynamically

👉 Structure of a node:

[Data | Next]

Basic C++ Syntax (Node)

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

Examples

1. 10 → 20 → 30 → NULL
2. 5 → 15 → 25 → 35 → NULL
3. 100 → NULL
4. Empty list (NULL)
5. 1 → 2 → 3 → 4 → 5 → NULL

2. Implementing a Singly Linked List

Explanation

We create:

- Node structure
- Head pointer (starting point)
- Functions to manage the list

Syntax

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

Node* head = NULL;
```

Example Implementation

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

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

Node* head = NULL;

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

Examples

1. Create empty list
2. Add one node manually
3. Traverse list
4. Print list
5. Check if list is empty

3. Insertion to the Front of a Singly Linked List

Explanation

We insert a new node at the beginning:

1. Create new node
2. Point it to current head
3. Update head

Syntax

```
void insertFront(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->next = head;
   head = newNode;
}
```

Examples

1. Insert 10 → 10 → NULL
2. Insert 20 → 20 → 10 → NULL
3. Insert 5 → 5 → 20 → 10
4. Insert 100 → 100 → ...
5. Insert into empty list

4. Removal from the Front of a Singly Linked List

Explanation

Remove first node:

1. Check if empty
2. Store head
3. Move head to next
4. Delete old node

Syntax

```
void deleteFront() {
   if(head == NULL) return;

   Node* temp = head;
   head = head->next;
   delete temp;
}
```

Examples

1. Remove from 10 → 20 → 30
2. Remove from single node
3. Remove until empty

4. Remove from empty list
5. Remove repeatedly

5. Implementing a Generic Singly Linked List

Explanation

Generic means **works for any data type** (int, float, string)

We use **templates** in C++.

Syntax

```
template <typename T>
struct Node {
   T data;
   Node* next;
};
```

Example

```
Node<int>* headInt;
Node<string>* headStr;
```

Examples

1. Integer list
2. Float list
3. String list
4. Char list
5. Double list

6. Doubly Linked Lists

Explanation

Each node has:

- Data
- Pointer to next
- Pointer to previous

👉 Structure:

[Prev | Data | Next]

Syntax

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

Examples

1. NULL ← 10 ⇄ 20 ⇄ 30 → NULL
2. Single node
3. Forward traversal
4. Backward traversal
5. Empty list

7. Insertion into a Doubly Linked List

Explanation

Steps:

1. Create node
2. Adjust next and prev pointers

Syntax (Insert at front)

```
void insertFront(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->prev = NULL;
   newNode->next = head;

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

   head = newNode;
}
```

Examples

1. Insert in empty list
2. Insert at front
3. Insert multiple nodes
4. Insert between nodes
5. Insert at end

8. Removal from a Doubly Linked List

Explanation

We must update both:

- next pointer
- prev pointer

Syntax (Delete front)

```
void deleteFront() {
   if(head == NULL) return;

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

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

   delete temp;
}
```

Examples

1. Remove first node
2. Remove last node
3. Remove middle node
4. Remove single node
5. Remove all nodes

9. Full C++ Implementation

Complete Example

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

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

Node* head = NULL;

void insertFront(int value) {
   Node* newNode = new Node();
   newNode->data = value;
   newNode->next = head;
   head = newNode;
```

```
}

void deleteFront() {
   if(head == NULL) return;

   Node* temp = head;
   head = head->next;
   delete temp;
}

void display() {
   Node* temp = head;
   while(temp != NULL) {
      cout << temp->data << " -> ";
      temp = temp->next;
   }
   cout << "NULL\n";
}

int main() {
   insertFront(10);
   insertFront(20);
   insertFront(30);

   display();

   deleteFront();
   display();

   return 0;
}
```

Examples

1. Insert and display
2. Insert multiple values
3. Delete and display
4. Work with empty list
5. Continuous insert/delete

Final Notes:

- Linked Lists are **dynamic** (flexible size)
- They use **pointers**
- Slower than arrays for access (no indexing)
- Very useful for:
    - Memory management
    - Dynamic data
    - Implementing stacks/queues