LIINKED LIST

1️⃣ Introduction & Basics

Definition:

Linked List is a linear data structure where nodes are connected using pointers. Unlike arrays, memory is **non-contiguous**.

Node Structure (Singly LL):

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

Types:

1. Singly Linked List (SLL)
2. Doubly Linked List (DLL)
3. Circular Linked List (CLL)
4. Doubly Circular Linked List

Why use LL over Array?

- Dynamic size
- Easy insertion/deletion
- No memory wastage

2️⃣ Basic Operations

2.1 Traversal

```
void traverse(Node* head){
    Node* temp = head;
    while(temp){
```

```
        cout << temp->data << " ";
        temp = temp->next;
    }
}
```

2.2 Insertion

At Head

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

At Tail

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

At Position

- Traverse to `pos-1` node
- Update `next` pointers

2.3 Deletion

At Head

```
void deleteHead(Node*& head){
    if(!head) return;
    Node* temp = head;
    head = head->next;
```

```
    delete temp;
}
```

At Tail / Specific Position

- Use prev pointer
- Update `prev->next = curr->next`
- Delete `curr`

3️⃣ Patterns in Linked List (Must Know)

Pattern 1: Slow & Fast Pointer

Use Cases: Middle, Cycle detection, Palindrome

Middle Node

```
Node* middle(Node* head){
    Node* slow = head;
    Node* fast = head;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}
```

Detect Cycle

```
bool hasCycle(Node* head){
    Node *slow = head, *fast = head;
    while(fast && fast->next){
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast) return true;
    }
```

```
    return false;
}
```

Remove Cycle

- Find meeting point → Move one ptr to head
- Move both 1 step → meet at cycle start
- Set `prev->next = nullptr`

Pattern 2: Reverse Linked List

Iterative

```
Node* reverse(Node* head){
    Node* prev = nullptr;
    Node* curr = head;
    Node* next;
    while(curr){
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

Recursive

```
Node* reverseRec(Node* head){
    if(!head || !head->next) return head;
    Node* newHead = reverseRec(head->next);
    head->next->next = head;
    head->next = nullptr;
    return newHead;
}
```

Use Cases:

- Palindrome check
- Reverse part of list
- Merge sort on LL

Pattern 3: Merge Two Sorted Lists

```
Node* merge(Node* l1, Node* l2){
    if(!l1) return l2;
    if(!l2) return l1;
    if(l1->data < l2->data){
        l1->next = merge(l1->next, l2);
        return l1;
    } else {
        l2->next = merge(l1, l2->next);
        return l2;
    }
}
```

Pattern 4: Remove N-th Node From End

- Use **dummy node**
- Move fast pointer n+1 steps
- Move slow + fast together → remove `slow->next`

```
Node* removeNth(Node* head, int n){
    Node* dummy = new Node{0, head};
    Node* slow = dummy, *fast = dummy;
    for(int i=0;i<=n;i++) fast = fast->next;
    while(fast){
        slow = slow->next;
        fast = fast->next;
    }
```

```
    Node* toDelete = slow→next;
    slow→next = slow→next→next;
    delete toDelete;
    return dummy→next;
}
```

Pattern 5: Linked List Palindrome

1. Find middle
2. Reverse second half
3. Compare first half & reversed half
4. Optional: restore list

```
bool isPalindrome(Node* head){
    if(!head) return true;
    Node* mid = middle(head);
    Node* rev = reverse(mid→next);
    Node* p1 = head, *p2 = rev;
    while(p2){
        if(p1→data != p2→data) return false;
        p1 = p1→next;
        p2 = p2→next;
    }
    mid→next = reverse(rev); // restore
    return true;
}
```

Pattern 6: Intersection of Two Lists

1. Find lengths
2. Move longer list pointer difference steps
3. Move both → first intersect node

```
Node* getIntersection(Node* A, Node* B){
    int lenA=0,lenB=0;
    Node* temp = A;
    while(temp){ lenA++; temp=temp→next; }
    temp = B;
    while(temp){ lenB++; temp=temp→next; }

    while(lenA>lenB){ A=A→next; lenA--; }
    while(lenB>lenA){ B=B→next; lenB--; }

    while(A && B){
        if(A==B) return A;
        A=A→next;
        B=B→next;
    }
    return nullptr;
}
```

Pattern 7: Reorder List / Swap Nodes

- Reverse second half → Merge alternate nodes
- Swap pairs → Update `next` pointers

Pattern 8: Advanced – Flatten Multilevel List

- Recursively attach `child` lists to next nodes
- Use recursion or stack

5️⃣ Quick Tips & Tricks

1. Use **dummy node** for complex insert/delete
2. Always check **edge cases**:
    - Empty list
    - Single node

- Two nodes

3. **Slow-Fast pointers** → remember slow=1 step, fast=2 steps
4. **Reversing** → track prev, curr, next
5. **Merging lists** → recursion cleaner, iteration faster
6. Draw pointers on paper → visualize before coding