Complete Linked List Pattern - Deep Study Guide

Table of Contents

---

1. Theoretical Foundation

What is a Linked List?

Definition: A linear data structure where elements (nodes) are stored in non-contiguous memory locations, with each node containing data and pointer(s) to the next (and/or previous) node.

Historical Context:

- Invented by Allen Newell, Cliff Shaw, and Herbert A. Simon (1955-1956)
- First used in the Information Processing Language (IPL)
- Foundation for dynamic memory allocation
- Critical for understanding pointer manipulation in systems programming

Core Principle: Trade random access (O(1)) for dynamic size and efficient insertion/deletion at known positions.

Memory Architecture Deep Dive

Arrays vs Linked Lists - Memory Layout

```
ARRAY IN MEMORY:
Address: 1000  1004  1008  1012  1016
Value:   [10]  [20]  [30]  [40]  [50]
         ↑ Contiguous block
```

```
Advantages:
- Cache-friendly (spatial locality)
- O(1) random access
- Less memory per element

Disadvantages:
- Fixed size or expensive reallocation
- O(n) insertion/deletion in middle
- Memory fragmentation if large


LINKED LIST IN MEMORY:
Node A @ 1000: {data: 10, next: 2500}
Node B @ 2500: {data: 20, next: 1200}
Node C @ 1200: {data: 30, next: 3000}
Node D @ 3000: {data: 40, next: NULL}
        ↑ Scattered across memory

Advantages:
- Dynamic size
- O(1) insertion/deletion at known position
- No reallocation needed

Disadvantages:
- Cache-unfriendly
- O(n) access
- Extra memory for pointers
```

Pointer Mechanics - The Real Truth

cpp

```cpp
// What happens in memory?
ListNode* ptr = new ListNode(10);

// ptr is a VARIABLE storing an ADDRESS
// Let's say new allocates at address 0x1000

ptr      → 0x1000 (address stored in ptr)
*ptr     → Node{val: 10, next: nullptr} (object at 0x1000)
ptr->val → 10 (member access via pointer)
&ptr     → 0x7FFE (address of ptr itself on stack)

// Critical Understanding
ListNode* a = new ListNode(5);
ListNode* b = a;  // b points to SAME node as a

b->val = 10;  // Changes a->val too! They share the node.

// This is different from:
ListNode* c = new ListNode(a->val);  // NEW node with copied value
c->val = 10;  // a->val still 5
```

Memory Leak - The Silent Killer

cpp

```cpp
// LEAK EXAMPLE
void createLeak() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);
    // Function ends, 'head' goes out of scope
    // Nodes still in heap, address lost
    // Memory leaked! ❌
}

// CORRECT - Manual Cleanup
void properCleanup() {
    ListNode* head = new ListNode(1);
    head->next = new ListNode(2);

    // Must delete before returning
    while (head != nullptr) {
        ListNode* temp = head;
        head = head->next;
        delete temp;  // Free memory
    }
}

// BEST - Smart Pointers (C++11+)
void modernWay() {
    unique_ptr<ListNode> head = make_unique<ListNode>(1);
    head->next = make_unique<ListNode>(2);
    // Automatically freed when out of scope ✓
}
```

Why Linked Lists Matter

Real-World Applications:

1. **Operating Systems:** Process scheduling queues
2. **Browsers:** Forward/backward navigation (doubly linked)
3. **Music Players:** Playlist management (circular linked)
4. **Undo/Redo:** Command pattern implementation
5. **Hash Tables:** Chaining for collision resolution
6. **LRU Cache:** Combines hash map + doubly linked list
7. **Graph Adjacency:** Adjacency list representation

When to Use:

- ✅ Frequent insertions/deletions (especially at beginning)
- ✅ Don't know size in advance
- ✅ Need to split/merge sequences efficiently
- ✅ Implementing stacks, queues, deques
- ❌ Need random access (use array/vector)
- ❌ Memory is extremely limited (pointer overhead)
- ❌ Cache performance critical (use array)

---

2. Mathematical Analysis

Complexity Analysis - Detailed Proofs

Insertion Complexity

At Head - O(1):

```
Proof:
1. Create new node: O(1)
2. Set newNode->next = head: O(1)
3. Update head = newNode: O(1)
Total: O(1) - constant operations regardless of list size
```

At Tail - O(n) without tail pointer:

```
Proof:
1. Traverse to last node: O(n)
   - Must visit n-1 nodes
   - Each visit is O(1)
   - Total: (n-1) × O(1) = O(n)
2. Create new node: O(1)
3. Link last->next = newNode: O(1)
Total: O(n) + O(1) = O(n)

With tail pointer: O(1)
```

At Position k - O(k):

```
Proof:
1. Traverse to position k-1: O(k)
2. Update pointers: O(1)
Total: O(k) - linear in position
```

Search Complexity - O(n)

```
Mathematical Proof:
Best case: Element at head → O(1)
Average case: Element at middle → O(n/2) = O(n)
Worst case: Element at end or not present → O(n)

Expected value:
E[comparisons] = Σ(i=1 to n) (i × P(position=i))
             = Σ(i=1 to n) (i × 1/n)
             = (1/n) × Σ(i=1 to n) i
             = (1/n) × n(n+1)/2
             = (n+1)/2
             = O(n)
```

Space Complexity Analysis

```
Per node overhead:
- Singly: 1 pointer (8 bytes on 64-bit systems)
- Doubly: 2 pointers (16 bytes)
- Data: Depends on data type

Example: Storing n integers
Array:     4n bytes
Singly LL: 4n + 8n = 12n bytes (3x overhead)
Doubly LL: 4n + 16n = 20n bytes (5x overhead)

Trade-off: Memory overhead for flexibility
```

Comparison with Other Structures

Operation	Array	Singly LL	Doubly LL	Skip List
Access ith	O(1)	O(i)	O(min(i, n-i))	O(log n)
Search	O(n)	O(n)	O(n)	O(log n)
Insert Head	O(n)	O(1)	O(1)	O(log n)

Operation	Array	Singly LL	Doubly LL	Skip List
Insert Tail	O(1)*	O(n)/O(1)**	O(1)	O(log n)
Insert Middle	O(n)	O(1)***	O(1)***	O(log n)
Delete Head	O(n)	O(1)	O(1)	O(log n)
Delete Tail	O(1)	O(n)	O(1)	O(log n)
Delete Middle	O(n)	O(1)***	O(1)***	O(log n)
Space per element	4-8 bytes	12-16 bytes	20-24 bytes	16-24 bytes

* Amortized
** Without/with tail pointer
*** If node pointer is known

---

3. Six Core Patterns - Deep Dive

Pattern 1: Two Pointer (Fast & Slow) - Floyd's Algorithm

Concept

Two pointers traverse list at different speeds to detect cycles, find middle, or locate specific positions.

Mathematical Foundation

Cycle Detection Proof:

```
Setup:
- Slow pointer moves 1 step per iteration
- Fast pointer moves 2 steps per iteration
- List has cycle of length C

Theorem: If cycle exists, fast and slow MUST meet.

Proof:
1. Let D = distance from head to cycle entrance
2. Slow enters cycle after D steps
3. At this point, fast is already in cycle (or enters shortly after)
4. Once both in cycle, analyze relative motion:

   In each iteration:
   - Fast moves 2 steps
   - Slow moves 1 step
   - Relative speed = 2 - 1 = 1 step/iteration

5. Gap between them decreases by 1 each iteration
6. Since cycle is finite (length C), gap must become 0
7. Maximum iterations to meet: C (one full cycle)

Time Complexity: O(n)
- If no cycle: Fast reaches end in n/2 steps
- If cycle: Meet within C steps after slow enters
- Total: O(D + C) ≤ O(n)

Space Complexity: O(1)
- Only 2 pointers
```

Finding Cycle Start - Mathematical Derivation:

```
Let:
- L = distance from head to cycle start
- k = distance from cycle start to meeting point
- C = cycle length
- m = number of complete cycles fast made

When they meet:
Distance traveled by slow = L + k
Distance traveled by fast = L + mC + k (m complete cycles)

Since fast travels 2× slow's distance:
2(L + k) = L + mC + k
2L + 2k = L + mC + k
L + k = mC
L = mC - k

This means:
L = (m-1)C + (C - k)

Where (C - k) = distance from meeting point back to cycle start

Conclusion:
Distance from head to cycle start =
Distance from meeting point to cycle start

Algorithm:
1. Detect cycle, find meeting point
2. Reset one pointer to head
3. Move both at speed 1
4. They meet at cycle start

Proof of correctness: Both travel distance L to cycle start
```

Implementation Pattern

```
cpp
```

```
// Template for Fast-Slow Pattern
ListNode* fastSlowPattern(ListNode* head) {
    if (head == nullptr) return nullptr;

    ListNode* slow = head;
    ListNode* fast = head;

    // Phase 1: Move pointers
    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;          // 1 step
        fast = fast->next->next;    // 2 steps

        // Check condition (varies by problem)
        if (slow == fast) {
            // Cycle detected or specific position found
            break;
        }
    }

    // Phase 2: Additional processing if needed
    // (e.g., finding cycle start)

    return result;
}
```

Variations & Applications

1. Find Middle Node

```
When fast reaches end:
- Fast traveled 2n steps (where n = slow's steps)
- Slow at position n
- For even length: slow at second middle
- For odd length: slow at exact middle

For first middle (even length):
Start fast at head->next instead of head
```

2. Find Nth from End

```
Create gap of n between fast and slow:
1. Move fast n steps ahead
2. Move both together until fast reaches end
3. Slow now at nth from end

Proof:
Gap = n nodes
When fast at end (position n from end is head)
Slow is n positions behind = nth from end
```

3. Check Palindrome

```
Combine multiple patterns:
1. Fast-slow to find middle
2. Reverse second half
3. Compare both halves

Time: O(n), Space: O(1)
```

Pattern 2: Reversal - Pointer Manipulation

Concept

Change direction of pointers to reverse list or parts of it.

Mathematical Analysis

Iterative Reversal:

```
State Machine Analysis:

State at iteration i:
- prev: Points to reversed portion's head (or null initially)
- curr: Node being processed
- next: Saved reference to rest of list

Invariant:
After iteration i:
- Nodes 0 to i-1 are reversed
- curr points to node i
- prev points to node i-1
- List from i onwards is unchanged

Initial state (i=0):
prev = null, curr = head

Transition (process node i):
1. next = curr->next        [Save future]
2. curr->next = prev        [Reverse pointer]
3. prev = curr              [Move reversed head]
4. curr = next              [Move to next]

Termination (curr == null):
prev points to new head

Time: O(n) - visit each node once
Space: O(1) - only 3 pointers
```

Recursive Reversal:

```
Recurrence Relation:

reverse(head) = {
    head                          if head == null or head->next == null
    reverse(head->next) + adjust   otherwise
}

"adjust" means:
head->next->next = head
head->next = null

Proof of Correctness by Induction:

Base case (n=1): Single node, return as is ✓

Inductive hypothesis: reverse(list of n-1) works correctly

Inductive step: For list of n nodes
1. Recursively reverse last n-1 nodes
2. First node now at end
3. Adjust: last node's next points back to first
4. First node's next becomes null
Result: Entire list reversed ✓

Time: O(n) - T(n) = T(n-1) + O(1) = O(n)
Space: O(n) - recursion stack depth
```

Implementation Pattern

```cpp
```

```cpp
// Iterative Template (Optimal)
ListNode* reverseIterative(ListNode* head) {
    ListNode* prev = nullptr;
    ListNode* curr = head;

    while (curr != nullptr) {
        ListNode* next = curr->next;  // Save
        curr->next = prev;            // Reverse
        prev = curr;                  // Advance
        curr = next;
    }

    return prev;  // New head
}

// Recursive Template (Elegant)
ListNode* reverseRecursive(ListNode* head) {
    // Base case
    if (head == nullptr || head->next == nullptr) {
        return head;
    }

    // Recursive case
    ListNode* newHead = reverseRecursive(head->next);

    // Adjust pointers
    head->next->next = head;
    head->next = nullptr;

    return newHead;
}
```

Advanced: Reverse in Groups

Reverse K Nodes:

```
Problem: Reverse every k nodes

Analysis:
1. Check if k nodes available
2. Reverse k nodes
3. Recursively handle rest
4. Connect pieces

Complexity:
Time: T(n) = T(n-k) + O(k) = O(n)
- Each node processed once
- Checking k nodes: O(k) per group
- n/k groups → O(n)

Space: O(n/k) for recursion
- Iterative version: O(1)
```

Reverse Between Positions [m, n]:

```
Approach:
1. Traverse to position m-1
2. Reverse nodes from m to n
3. Adjust connections

Key insight: Use dummy node to handle m=1 case

Time: O(n)
Space: O(1)
```

Pattern 3: Dummy Node - Simplification Technique

Concept

Create artificial head node to eliminate special cases.

Why Dummy Nodes?

Problem Without Dummy:

```cpp
```

```cpp
// Deleting head requires special handling
if (nodeToDelete == head) {
    head = head->next;  // Special case
    delete nodeToDelete;
} else {
    // Normal case
    prev->next = nodeToDelete->next;
    delete nodeToDelete;
}
```

Solution With Dummy:

```cpp
ListNode* dummy = new ListNode(0);
dummy->next = head;

// Now head is just another node!
// No special case needed
prev->next = nodeToDelete->next;
delete nodeToDelete;

return dummy->next;  // New head (might be different)
```

Mathematical Justification

```
Edge Cases in Linked List Operations:

Without dummy node:
- Empty list: 1 special case
- Single node: 1 special case
- Operation on head: 1 special case
- Normal case: 1 case
Total: 4 code paths

With dummy node:
- Dummy->next = head (even if null)
- All operations on dummy->next onwards
- Return dummy->next
Total: 1 unified code path

Code complexity reduction: 4 → 1
Bug probability reduction: ~75%
```

Implementation Pattern

```cpp
// Template with Dummy Node
ListNode* processWithDummy(ListNode* head) {
    ListNode* dummy = new ListNode(0);
    dummy->next = head;

    ListNode* prev = dummy;
    ListNode* curr = head;

    while (curr != nullptr) {
        if (shouldProcess(curr)) {
            // Modify list
            prev->next = curr->next;
            // Don't move prev
        } else {
            prev = curr;
        }
        curr = curr->next;
    }

    ListNode* newHead = dummy->next;
    delete dummy;  // Clean up
    return newHead;
}
```

Applications

1. Remove Elements:

- No special handling for head removal
- Unified deletion logic

2. Partition List:

- Create two dummy nodes for two partitions
- Simpler merging

3. Merge Lists:

- One dummy for result
- Clean iteration logic

Pattern 4: In-Place Pointer Manipulation

Concept

Rearrange nodes by changing pointers without creating new nodes.

Memory Efficiency Analysis

```
Creating new list:
- Allocate n new nodes: O(n) space
- Copy values: O(n) time
- Total: O(n) time, O(n) space

In-place manipulation:
- Only change pointers: O(n) time
- No new allocations: O(1) space
- Total: O(n) time, O(1) space

Space savings: n × sizeof(Node)
```

Classic Examples

1. Swap Nodes in Pairs:

```
Before: 1 → 2 → 3 → 4
After:  2 → 1 → 4 → 3

Pointer changes per pair:
- first->next = second->next
- second->next = first
- prev->next = second

All O(1) operations
```

2. Odd-Even List:

```
Before: 1 → 2 → 3 → 4 → 5
After:  1 → 3 → 5 → 2 → 4

Algorithm:
1. Separate into odd and even lists
2. Connect odd tail to even head

Time: O(n) - single pass
Space: O(1) - only 4 pointers (odd, even, oddHead, evenHead)
```

3. Reorder List:

```
Before: L0 → L1 → L2 → L3 → L4
After:  L0 → L4 → L1 → L3 → L2

Multi-pattern combination:
1. Find middle (fast-slow): O(n/2)
2. Reverse second half: O(n/2)
3. Merge alternately: O(n/2)
Total: O(n), Space: O(1)
```

Pattern 5: Merge Technique

Concept

Combine two or more sorted lists into one sorted list.

Merge Two Sorted Lists - Analysis

```
Lists: L1 (m nodes), L2 (n nodes)

Algorithm:
1. Compare L1[i] and L2[j]
2. Take smaller, advance that pointer
3. Append remaining

Time Complexity Proof:
- Each comparison advances one pointer
- Total advancements: m + n
- Each advancement: O(1)
- Total: O(m + n)

Space Complexity:
- Reusing input nodes: O(1)
- Creating new list: O(m + n)
```

Merge K Sorted Lists - Deep Analysis

Approach 1: Sequential Merging

```
Merge list1 with list2 → temp1
Merge temp1 with list3 → temp2
...
Merge temp(k-2) with listk → result

Time Analysis:
First merge: O(n₁ + n₂)
Second merge: O(n₁ + n₂ + n₃)
...
Kth merge: O(n₁ + n₂ + ... + nₖ)

Total: O(kN) where N = total nodes
Too slow! ❌
```

Approach 2: Min-Heap (Priority Queue)

```
1. Add first node of each list to min-heap: O(k log k)
2. While heap not empty:
   - Extract min: O(log k)
   - Add min's next to heap: O(log k)
3. Repeat N times

Total Time: O(N log k)
Space: O(k) for heap

Much better! ✓
```

Approach 3: Divide & Conquer

```
Pair up lists and merge:
Round 1: k lists → k/2 merged lists
Round 2: k/2 lists → k/4 merged lists
...
Round log k: 1 final list

Time per round: O(N) (merging all nodes)
Number of rounds: O(log k)
Total: O(N log k)

Space: O(log k) for recursion

Optimal! ✓✓
```

Comparison

Approach	Time	Space	Best For
Sequential	O(kN)	O(1)	k is very small (≤3)
Min-Heap	O(N log k)	O(k)	General case, clean code
Divide-Conquer	O(N log k)	O(log k)	Optimal, interview favorite

---

Pattern 6: Runner Technique (Advanced Two Pointer)

Concept

One pointer moves multiple steps for each step of another, not just 2×.

Applications

1. Weaving Lists:

```
Split list into two halves
Weave them together

Example: 1→2→3→4→5→6
Result:  1→4→2→5→3→6
```

2. Rotating List:

```
Find length while finding tail
Use modular arithmetic
Break and reconnect at rotation point

Time: O(n), Space: O(1)
```

3. K-th Group Operations:

```
Process every k-th node
Runner moves k steps
Worker processes
```

---

4. Advanced Pattern Recognition

Decision Framework

```
START: Analyze problem

Is it CYCLE related?
├──→ YES: Floyd's Algorithm (Fast-Slow)
│   └──→ Detect cycle, Find cycle start, Find cycle length
│
Is it REVERSAL related?
├──→ YES: Reversal Pattern
│   ├──→ Entire list: Iterative O(1) or Recursive O(n)
│   ├──→ Partial: Between positions, K groups
│   └──→ Consider: Iterative is always better for space
│
Does HEAD CHANGE frequently?
├──→ YES: Dummy Node Pattern
│   └──→ Remove elements, Partition, Merge operations
│
Need to REARRANGE nodes?
├──→ YES: In-Place Pointer Manipulation
│   └──→ Swap pairs, Odd-even, Reorder
│
MERGING lists?
├──→ YES: Merge Pattern
│   ├──→ 2 lists: Two pointer merge
│   ├──→ K lists: Heap or Divide-Conquer
│   └──→ Both O(N log k) optimal
│
Finding SPECIFIC POSITION?
├──→ YES: Runner Technique
│   ├──→ Middle: Fast-slow
│   ├──→ Nth from end: Gap of n
│   └──→ K-th element: Custom gap
│
MULTIPLE PATTERNS needed?
└──→ YES: Combine patterns systematically
    └──→ Example: Palindrome = Fast-slow + Reverse + Compare
```

Complexity Decision Tree

```
Space is CRITICAL?
├──→ YES: Prefer iterative, avoid hash maps
│   └──→ O(1) space solutions only
│
```

```
Need FAST?
├──→ YES: Optimize for time
│   └──→ Use hash maps, preprocessing if needed
│
Input MODIFIED allowed?
├──→ NO: Must copy or restore
│   └──→ Careful with reversal in palindrome
│
What's MORE IMPORTANT?
├──→ Code clarity: Recursive solutions
├──→ Performance: Iterative solutions
└──→ Interview: Start recursive, optimize to iterative
```

Pattern Combination Strategy

```
Complex problems often need 2-3 patterns:

Example: Copy List with Random Pointer
Solution 1: Hash Map
- Pattern: Traversal + Mapping
- Space: O(n)

Solution 2: Interweaving (O(1) space)
- Pattern 1: In-place manipulation (interweave)
- Pattern 2: Traversal (set random)
- Pattern 3: In-place separation
- Space: O(1)

Interview tip: Explain both, implement optimal
```

---

5. Problem Categories with Analysis

Category 1: Cycle Detection Problems

Theoretical Foundation

Why Cycles are Special:

```
In normal list: finite traversal
With cycle: infinite traversal possible
Challenge: Detect without infinite loop
Solution: Floyd's tortoise and hare
```

Problems & Analysis

1. Linked List Cycle (LC 141) ⭐ Foundation

- **Difficulty:** Easy
- **Pattern:** Fast-Slow (Floyd's)
- **Key Concept:** Meeting implies cycle
- **Time:** O(n), **Space:** O(1)

2. Linked List Cycle II (LC 142) ⭐ Mathematical

- **Difficulty:** Medium
- **Pattern:** Floyd's + Mathematical proof
- **Key Insight:** L = distance to start = distance from meet to start
- **Application:** Memory leak detection

3. Happy Number (LC 202)

- **Difficulty:** Easy
- **Pattern:** Floyd's on number sequence
- **Connection:** Sequence forms implicit linked list
- **Key Insight:** Cycle detection in transformation

4. Find Duplicate Number (LC 287) ⭐ Advanced

- **Difficulty:** Medium
- **Pattern:** Floyd's on array as linked list
- **Constraint:** Can't modify array, O(1) space
- **Key Mapping:** arr[i] = next pointer

5. Remove Cycle

- **Difficulty:** Medium
- **Pattern:** Floyd's + Pointer adjustment
- **Challenge:** Break cycle at correct point
- **Edge Case:** Cycle starts at head

6. Cycle Length Calculation

- **Difficulty:** Easy
- **Pattern:** Floyd's + Counting
- **Method:** Count steps from meeting to meeting again

7. Check if Linked List is Circular

- **Difficulty:** Easy
- **Pattern:** Modified Floyd's
- **Difference:** Head in cycle vs separate cycle

Category 2: Reversal Problems

Problem Progression (Easy → Hard)

8. Reverse Linked List (LC 206) ⭐ ⭐ Must Master

- **Difficulty:** Easy
- **Pattern:** Basic reversal (iterative & recursive)
- **Foundation:** All reversal problems build on this
- **Both approaches required:** Know iterative (O(1)) and recursive (O(n))

9. Reverse Linked List II (LC 92)

- **Difficulty:** Medium
- **Pattern:** Partial reversal
- **Challenge:** Maintain connections at boundaries
- **Key Insight:** Use dummy node for m=1 case

10. Reverse Nodes in K-Group (LC 25) ⭐ Hard Classic

- **Difficulty:** Hard
- **Pattern:** Reversal + Recursion/Iteration
- **Challenge:** Only reverse if k nodes available
- **Interview Favorite:** Tests multiple skills

11. Reverse Alternate K Nodes

- **Difficulty:** Medium

- **Pattern:** Conditional reversal
- **Approach:** Reverse k, skip k, repeat

12. Swap Nodes in Pairs (LC 24)

- **Difficulty:** Medium
- **Pattern:** K=2 reversal
- **Special Case:** Of reverse K groups

13. Reverse in Place with Constraint

- **Difficulty:** Hard
- **Constraint:** Only certain nodes can be reversed
- **Variation:** Reverse only even/odd valued nodes

Category 3: Middle & Partition Problems

14. Middle of Linked List (LC 876) ⭐ Foundation

- **Difficulty:** Easy
- **Pattern:** Fast-slow
- **Variations:** First middle vs second middle
- **Application:** Used in many complex problems

15. Delete Middle Node (LC 2095)

- **Difficulty:** Medium
- **Pattern:** Fast-slow + deletion
- **Challenge:** Maintain pointer to previous

16. Partition List (LC 86)

- **Difficulty:** Medium
- **Pattern:** Two dummy nodes
- **Approach:** Separate into two lists, merge
- **Application:** Quicksort on linked list

17. Odd Even Linked List (LC 328)

- **Difficulty:** Medium
- **Pattern:** Two pointers + separation
- **Key Insight:** In-place rearrangement

18. Sort List (LC 148) ⭐ Important

- **Difficulty:** Medium
- **Pattern:** Merge sort (find middle + merge)
- **Optimal:** O(n log n) time, O(log n) space
- **Interview Favorite:** Complete algorithm design

19. Insertion Sort List (LC 147)

- **Difficulty:** Medium
- **Pattern:** Sorted insertion with traversal
- **Time:** $O(n^2)$ but demonstrates insertion technique

Category 4: Merge & Combine Problems

20. Merge Two Sorted Lists (LC 21) ⭐ Foundation

- **Difficulty:** Easy
- **Pattern:** Basic merge
- **Must Master:** Foundation for merge k lists

21. Merge K Sorted Lists (LC 23) ⭐ ⭐ Hard Classic

- **Difficulty:** Hard
- **Pattern:** Heap OR Divide-Conquer
- **Two Solutions:** Both O(N log k)
- **Interview:** Explain trade-offs

22. Merge In Between Lists (LC 1669)

- **Difficulty:** Medium
- **Pattern:** Merge + traversal
- **Challenge:** Find positions, connect properly

23. Add Two Numbers (LC 2)

- **Difficulty:** Medium
- **Pattern:** Merge + carry handling
- **Key:** Process simultaneously with carry

24. Add Two Numbers II (LC 445)

- **Difficulty:** Medium
- **Pattern:** Reverse + Add + Reverse
- **Challenge:** Most significant digit first

25. Multiply Two Numbers (Custom)

- **Difficulty:** Hard
- **Pattern:** Nested traversal + carry
- **Application:** BigInteger implementation

Category 5: Palindrome Problems

26. Palindrome Linked List (LC 234) ⭐ Multi-Pattern

- **Difficulty:** Easy
- **Patterns:** Fast-slow + Reverse + Compare
- **Challenge:** Combine 3 patterns correctly
- **Follow-up:** Restore original list?

27. Valid Palindrome (String - Related)

- **Difficulty:** Easy
- **Connection:** Two pointer technique
- **Application:** Understanding comparison logic

Category 6: Remove & Delete Problems

28. Remove Linked List Elements (LC 203)

- **Difficulty:** Easy

- **Pattern:** Dummy node + traversal
- **Key:** Handle multiple consecutive removals

29. Remove Duplicates from Sorted List (LC 83)

- **Difficulty:** Easy
- **Pattern:** Single pointer traversal
- **Approach:** Keep first occurrence

30. Remove Duplicates from Sorted List II (LC 82)

- **Difficulty:** Medium
- **Pattern:** Dummy + skip all duplicates
- **Challenge:** Remove ALL occurrences

31. Remove Nth Node from End (LC 19) ⭐ Classic

- **Difficulty:** Medium
- **Pattern:** Gap pointers (n+1 gap)
- **Interview Favorite:** Tests pointer arithmetic

32. Delete Node in Linked List (LC 237)

- **Difficulty:** Easy
- **Pattern:** Copy next, delete next
- **Constraint:** Node pointer given, not head
- **Trick Question:** Can't actually delete given node!

33. Remove Zero Sum Consecutive Nodes (LC 1171)

- **Difficulty:** Medium
- **Pattern:** Prefix sum + hash map
- **Advanced:** Multiple passes may be needed

Category 7: Intersection & Common Elements

34. Intersection of Two Linked Lists (LC 160) ⭐ Clever

- **Difficulty:** Easy
- **Pattern:** Length adjustment OR two pointer switch
- **Key Insight:** Equal path lengths after switch
- **Mathematical:** Path A+B = Path B+A

35. Check if Linked Lists are Identical

- **Difficulty:** Easy
- **Pattern:** Simultaneous traversal
- **Compare:** Values and structure

Category 8: Copy & Clone Problems

36. Copy List with Random Pointer (LC 138) ⭐⭐ Important

- **Difficulty:** Medium
- **Pattern:** Hash map OR Interweaving
- **Solution 1:** Hash map O(n) space
- **Solution 2:** Interweaving O(1) space
- **Interview:** Explain both, implement optimal

37. Clone Graph (LC 133)

- **Difficulty:** Medium
- **Pattern:** DFS/BFS + hash map
- **Connection:** Similar to copy with random
- **Application:** Graph traversal

Category 9: Reordering & Rearrangement

38. Reorder List (LC 143) ⭐ Multi-Pattern

- **Difficulty:** Medium
- **Patterns:** Fast-slow + Reverse + Merge
- **Challenge:** L0→Ln→L1→Ln-1→L2→...

- **Application:** Combines 3 core patterns

39. Rotate List (LC 61)

- **Difficulty:** Medium
- **Pattern:** Find length + make circular + break
- **Key:** Use modular arithmetic

40. Swap Kth Node from Beginning and End (LC 1721)

- **Difficulty:** Medium
- **Pattern:** Find positions + swap values
- **Challenge:** Handle overlapping cases

41. Plus One Linked List (LC 369)

- **Difficulty:** Medium
- **Pattern:** Reverse + add + reverse OR stack
- **Challenge:** Carry propagation

Category 10: Special Structures

42. Flatten Multilevel Doubly Linked List (LC 430)

- **Difficulty:** Medium
- **Pattern:** DFS + pointer management
- **Challenge:** Maintain prev/next and child
- **Application:** Tree flattening

43. LRU Cache (LC 146) ⭐⭐⭐ System Design

- **Difficulty:** Medium
- **Pattern:** Doubly linked list + hash map
- **Must Know:** Classic interview question
- **Operations:** All O(1) - get, put, evict

44. LFU Cache (LC 460)

- **Difficulty:** Hard

- **Pattern:** Multiple DLLs + hash maps
- **More Complex:** Than LRU
- **Interview:** Usually only asked after LRU

45. Design Browser History (LC 1472)

- **Difficulty:** Medium
- **Pattern:** Doubly linked list
- **Operations:** Visit, back, forward

46. Design Skiplist (LC 1206)

- **Difficulty:** Hard
- **Pattern:** Multiple linked lists with levels
- **Advanced:** Probabilistic structure
- **Time:** O(log n) operations

---

6. 100+ Curated Problems

Easy Problems (30)

Foundation (Must Master - 10)

1. **Reverse Linked List (LC 206)** ⭐⭐
2. **Merge Two Sorted Lists (LC 21)** ⭐⭐
3. **Linked List Cycle (LC 141)** ⭐⭐
4. **Middle of Linked List (LC 876)** ⭐
5. **Remove Duplicates from Sorted List (LC 83)** ⭐
6. **Palindrome Linked List (LC 234)** ⭐
7. **Remove Linked List Elements (LC 203)** ⭐
8. **Intersection of Two Linked Lists (LC 160)** ⭐
9. **Delete Node in Linked List (LC 237)** ⭐
10. **Convert Binary Number in Linked List to Integer (LC 1290)**

Practice & Variations (20)

11. **Get Decimal Value of Binary Linked List**
12. **Find Length of Linked List**
13. **Find Nth Node from Beginning**
14. **Print Linked List in Reverse (Recursion)**
15. **Check if Linked List is Sorted**
16. **Insert in Sorted Linked List**
17. **Remove All Occurrences of Value**
18. **Find Max Element in List**
19. **Find Min Element in List**
20. **Sum of All Nodes**
21. **Product of All Nodes**
22. **Count Nodes in Linked List**
23. **Swap First and Last Nodes**
24. **Create Linked List from Array**
25. **Segregate Even and Odd Nodes**
26. **Happy Number (LC 202)**
27. **Is Subsequence (LC 392)** - Related concept
28. **Design Linked List (LC 707)**
29. **Reverse Print Linked List**
30. **Find Last Occurrence of Value**

Medium Problems (50)

Two Pointer & Fast-Slow (10)

31. **Linked List Cycle II (LC 142)** ⭐ ⭐
32. **Remove Nth Node from End (LC 19)** ⭐ ⭐
33. **Reorder List (LC 143)** ⭐ ⭐
34. **Rotate List (LC 61)** ⭐
35. **Delete Middle Node (LC 2095)**

36. **Maximum Twin Sum in Linked List (LC 2130)**
37. **Swap Nodes in Pairs (LC 24)**
38. **Odd Even Linked List (LC 328)**
39. **Split Linked List in Parts (LC 725)**
40. **Swap Kth Node from Beginning and End (LC 1721)**

Reversal Variants (8)

41. **Reverse Linked List II (LC 92)** ⭐ ⭐
42. **Add Two Numbers (LC 2)** ⭐
43. **Add Two Numbers II (LC 445)**
44. **Reverse Alternate K Nodes**
45. **Reverse Every Alternate M Nodes**
46. **Reverse Nodes Between (LC 92)**
47. **Double a Number Represented as Linked List (LC 2816)**
48. **Plus One Linked List (LC 369)**

Sorting & Partition (6)

49. **Sort List (LC 148)** ⭐ ⭐
50. **Insertion Sort List (LC 147)** ⭐
51. **Partition List (LC 86)** ⭐
52. **Sort Linked List of 0s 1s 2s**
53. **Segregate Even-Odd Valued Nodes**
54. **Sort Linked List Already Sorted Using Absolute Values**

Remove & Delete (7)

55. **Remove Duplicates from Sorted List II (LC 82)** ⭐
56. **Remove Zero Sum Consecutive Nodes (LC 1171)** ⭐
57. **Delete Nodes From Linked List Present in Array (LC 3217)**
58. **Delete Node Greater Than Value on Right**
59. **Delete N Nodes After M Nodes (LC 1474)**
60. **Delete Alternate Nodes**
61. **Remove Nodes with Same Value**

Merge & Combine (5)

62. **Merge In Between Linked Lists (LC 1669)**
63. **Merge Nodes in Between Zeros (LC 2181)**
64. **Add Two Polynomials Represented by Linked Lists**
65. **Subtract Two Numbers as Linked List**
66. **Merge K Sorted Lists (Prerequisite for hard)**

Complex Manipulation (8)

67. **Copy List with Random Pointer (LC 138)** ⭐⭐⭐
68. **Flatten Multilevel Doubly Linked List (LC 430)** ⭐
69. **Design Browser History (LC 1472)** ⭐
70. **Linked List Random Node (LC 382)**
71. **Next Greater Node in Linked List (LC 1019)**
72. **Linked List Components (LC 817)**
73. **Design Doubly Linked List**
74. **Implement Stack Using Linked List**

Special Applications (6)

75. **Implement Queue Using Linked List**
76. **Implement Deque Using Linked List**
77. **Design Circular Queue (LC 622)**
78. **Design Circular Deque (LC 641)**
79. **Design HashMap Using Chaining**
80. **Least Recently Used (LRU) - Medium version**

Hard Problems (20)

Advanced Algorithms (8)

81. **Reverse Nodes in K-Group (LC 25)** ⭐⭐⭐
82. **Merge K Sorted Lists (LC 23)** ⭐⭐⭐
83. **LRU Cache (LC 146)** ⭐⭐⭐

84. **LFU Cache (LC 460)** ⭐ ⭐
85. **All O'one Data Structure (LC 432)** ⭐
86. **Design Skiplist (LC 1206)** ⭐
87. **Find Duplicate Number (LC 287)** ⭐
88. **Reverse Alternate Nodes in K Groups**

System Design (6)

89. **Design Phone Directory (LC 379)**
90. **Insert Delete GetRandom O(1) (LC 380)**
91. **Design Twitter (LC 355)** - Uses linked list concepts
92. **Design In-Memory File System (LC 588)**
93. **Implement Trie Using Linked List**
94. **LRU Cache with Expiration Time**

Tree + Linked List (6)

95. **Flatten Binary Tree to Linked List (LC 114)**
96. **Convert Binary Search Tree to Sorted Doubly Linked List (LC 426)**
97. **Recover Binary Search Tree (LC 99)** - Linked list thinking
98. **Serialize and Deserialize Binary Tree (LC 297)**
99. **Linked List in Binary Tree (LC 1367)**
100. **Construct Binary Tree from Linked List**

Company-Specific Problems

Google (10)

- Merge K Sorted Lists
- LRU Cache
- Copy List with Random Pointer
- Reorder List
- Linked List Cycle II
- Add Two Numbers

- Remove Nth Node from End
- Sort List
- Flatten Multilevel Doubly Linked List
- Reverse Nodes in K-Group

Amazon (10)

- Reverse Linked List
- Merge Two Sorted Lists
- Remove Duplicates
- Linked List Cycle
- Intersection of Two Lists
- LRU Cache
- Copy List with Random Pointer
- Add Two Numbers
- Rotate List
- Partition List

Microsoft (10)

- Reverse Linked List
- Merge K Sorted Lists
- LRU Cache
- Remove Nth Node from End
- Reorder List
- Copy List with Random Pointer
- Design Browser History
- Sort List
- Add Two Numbers II
- Flatten Multilevel List

Facebook/Meta (10)

- Add Two Numbers
- Merge K Sorted Lists

- LRU Cache
- Copy List with Random Pointer
- Palindrome Linked List
- Reorder List
- Flatten Binary Tree to Linked List
- Linked List Random Node
- Design HashMap
- Intersection of Two Lists

Bloomberg (5)

- Reverse Linked List
- LRU Cache
- Merge Two Sorted Lists
- Add Two Numbers
- Remove Duplicates

Apple (5)

- Linked List Cycle
- Middle of Linked List
- Palindrome Linked List
- Merge Two Sorted Lists
- Reverse Linked List

---

7. Optimization Techniques

Technique 1: Space Optimization

Problem: Detect cycle with O(1) space

Naive Approach:

```
Use hash set to track visited nodes
Space: O(n)
```

Optimal Approach:

```
Floyd's algorithm
Space: O(1)
Key insight: Use speed difference, not storage
```

Problem: Copy list with random pointer O(1) space

Naive:

```
Hash map: old_node → new_node
Space: O(n)
```

Optimal:

```
Interweaving technique:
1. Create copies and interweave
2. Assign random pointers
3. Separate lists
Space: O(1)
```

Technique 2: Single Pass Algorithms

Problem: Find length AND middle in one pass

```cpp
```

```cpp
// Two operations in one traversal
pair<int, ListNode*> getLengthAndMiddle(ListNode* head) {
    int length = 0;
    ListNode* slow = head;
    ListNode* fast = head;

    while (fast != nullptr && fast->next != nullptr) {
        slow = slow->next;
        fast = fast->next->next;
        length += 2;
    }

    if (fast != nullptr) length++;

    return {length, slow};
}

// vs two passes:
// Pass 1: count length O(n)
// Pass 2: traverse to middle O(n/2)
// Single pass: O(n) but better constants
```

Technique 3: Dummy Node Mastery

When to Use Dummy Node

```
✓ Head might be removed
✓ List might become empty
✓ Multiple insertions at head
✓ Merging lists
✗ Only reading, not modifying
✗ Reversal (doesn't help)
✗ Cycle detection (not applicable)
```

Pattern

```cpp
```

```cpp
// Standard template
ListNode* dummy = new ListNode(0);
dummy->next = head;

// Process list
ListNode* curr = head;
ListNode* prev = dummy;

while (curr != nullptr) {
    // Modification logic
}

ListNode* newHead = dummy->next;
delete dummy;
return newHead;
```

Technique 4: Recursive Space Optimization

Tail Recursion Conversion

Regular Recursion (Not Tail):

cpp

```cpp
void printList(ListNode* head) {
    if (head == nullptr) return;
    cout << head->val << " ";
    printList(head->next);  // Not last operation
}
// Space: O(n) stack frames
```

Tail Recursion:

cpp

```cpp
void printListTail(ListNode* head) {
    if (head == nullptr) return;
    cout << head->val << " ";
    return printListTail(head->next);  // Last operation
}
// Compiler can optimize to O(1) space
```

Iterative (Best):

```cpp
void printListIterative(ListNode* head) {
    ListNode* curr = head;
    while (curr != nullptr) {
        cout << curr->val << " ";
        curr = curr->next;
    }
}
// Space: O(1) guaranteed
```

Technique 5: Preprocessing Strategies

Problem: Palindrome with restoration

```
Challenge: Check palindrome but restore original list

Naive:
1. Copy list: O(n) space
2. Check palindrome
3. Return original

Optimal:
1. Find middle
2. Reverse second half
3. Compare
4. Reverse second half again (restore)
5. Return result

Time: O(n), Space: O(1), List restored!
```

Technique 6: Mathematical Optimizations

Modular Arithmetic for Rotation

```
Problem: Rotate list by k positions

Naive:
for i = 0 to k:
    move last to first
```

```
Time: O(k × n) - very slow if k large!

Optimal:
k = k % length  // Key insight!
if k == 0: return head

Steps:
1. Find length and tail: O(n)
2. Connect tail to head (circular)
3. Find new tail at (length - k) position
4. Break circle

Time: O(n) even if k > n!
```

---

8. Study Notes & Mental Models

Mental Model 1: The Train Analogy

```
Think of linked list as train:

Head = Engine (knows where everything starts)
Nodes = Train cars (connected by couplers)
next pointer = Coupler (connects to next car)
nullptr = End of track

Operations:
- Add car at front: Attach to engine (O(1))
- Add car at end: Travel to end, attach (O(n))
- Remove car: Uncouple and reconnect (O(1) if position known)
- Reverse train: Flip all couplers (O(n))
- Find middle car: Fast train vs slow train race
```

Mental Model 2: The String of Pearls

```
Linked list = String with pearls

Each pearl = node
String segment = pointer

Breaking and reattaching:
- Careful not to lose pearls (memory leak!)
```

- Must hold both ends when breaking
- Can rearrange without recreating pearls
- If string breaks without holding: