Here are some of the most popular and widely used concepts and techniques for solving linked-list problems:

**Pointer Manipulation**

Understanding and manipulating pointers is crucial. Concepts like moving through the linked list using pointers, updating pointers, and swapping nodes are fundamental.

**Two Pointers Technique**

Using two pointers to traverse the linked list simultaneously can be useful in various scenarios, such as detecting cycles or finding the middle of the linked list.

**Dummy/Prehead Node**

Adding a dummy or prehead node before the actual head simplifies edge cases and avoids extra conditional checks when manipulating the list.

**Runner Technique**

In problems where you need to move at different speeds through the list (e.g., detecting cycles), using a slow and fast pointer (runner technique) can be effective.

**Recursion**

Solving linked-list problems recursively can lead to elegant solutions. Mastering recursion is crucial for many advanced problems.

**In-place Reversal**

Reversing a linked list in-place is a common operation. Understanding how to reverse a linked list efficiently is key.

**Cycle Detection**

Using Floyd's Tortoise and Hare algorithm for cycle detection is a common technique.

**Merge Sort for Linked Lists**

Understanding and implementing merge sort on linked lists is essential for problems that involve sorting or merging multiple lists.

**Intersection and Union of Linked Lists**

Solving problems related to finding the intersection or union of two linked lists requires careful traversal and manipulation.

**Stacks and Queues**

Utilizing stacks or queues can be helpful in solving certain linked-list problems, especially those involving reversal or managing elements in a particular order.

**Hashing**

Using hash tables can optimize certain linked-list problems, such as detecting duplicates or finding common elements in multiple lists.

**Sliding Window Technique**

Applicable in some linked-list problems where you need to maintain a subset of elements within a window.

**Flattening Nested Lists**

Understanding and implementing flattening algorithms for nested linked lists is crucial for some advanced problems.

**Priority Queue (Heap)**

In problems where maintaining a sorted order or finding the kth element is required, using a priority queue (heap) can be effective.

**Converting to Arrays**

Converting linked lists to arrays and solving problems in array format can sometimes simplify the solution.

Remember to practice regularly on platforms like LeetCode to reinforce these concepts and techniques. Each problem you solve will contribute to your understanding and proficiency in solving linked-list related problems.

Here's a list of widely used algorithms for solving linked list-related problems on LeetCode:

**Traversal**

Traverse the linked list from the beginning to the end, or vice versa, to perform various operations.

**Two Pointers**

Use two pointers to traverse the list at different speeds, often for finding the middle, detecting cycles, or solving intersection problems.

**In-place Reversal**

Reverse the linked list in-place by adjusting the pointers. This is a common technique for many problems.

**Merge Sort**

Implement the merge sort algorithm on linked lists for problems that involve sorting or merging multiple lists.

**Cycle Detection (Floyd's Tortoise and Hare)**

Use two pointers moving at different speeds to detect cycles in a linked list

**Intersection of Two Linked Lists**

Determine the intersection point of two linked lists using various techniques, such as calculating the difference in lengths or connecting the ends to form a cycle.

**Detecting Palindromes**

Check if a linked list is a palindrome using techniques like reversing the second half and comparing it with the first half.

**Remove Duplicates**

Remove duplicates from a sorted or unsorted linked list using various approaches, such as using a set or adjusting pointers.

**Swapping Nodes**

Swap nodes in a linked list, either adjacent nodes or nodes at given positions.

**Flattening Nested Lists**

Flatten a nested linked list structure by merging multiple linked lists into a single list.

**Kth to Last Element**

Find the kth to the last element in a linked list using two pointers or calculating the length.

**Partitioning**

Partition a linked list around a given value, similar to the partition step in quicksort.

**Addition of Two Numbers (Represented as Linked Lists)**

Perform addition on two numbers represented as linked lists.

**Clone a Linked List with Random Pointers**

Create a deep copy of a linked list that has random pointers in addition to the next pointers.

**Detecting and Removing Cycles**

Detect cycles in a linked list and remove them, often using Floyd's Tortoise and Hare algorithm.

**Odd-Even Linked List**

Rearrange a linked list such that all odd-indexed nodes come before the even-indexed nodes.

**Reorder Linked List**

Reorder a linked list to alternate between the first and last nodes, or some other specified pattern.

**LRU Cache using Linked List**

Implement an LRU (Least Recently Used) cache using a combination of a linked list and a hash table.

**Intersection of Three Sorted Linked Lists**

Find the common elements in three sorted linked lists.

**Convert Sorted List to Binary Search Tree**

Convert a sorted linked list to a balanced binary search tree.

These algorithms cover a wide range of linked list-related problems on LeetCode, and mastering them will enhance your problem-solving skills in this domain.