Sheryians Coding School

Live Cohort

Day 101

Class 1: Merging & Reversing Linked Lists

Topic: Merge Two Sorted Lists (Recursion)

Definition:

Merging two sorted linked lists into one sorted linked list by comparing elements recursively.

Code Example:

```
function mergeTwoLists(l1, l2) {
    if (!l1) return l2;
    if (!l2) return l1;

    if (l1.val < l2.val) {
        l1.next = mergeTwoLists(l1.next, l2);
        return l1;
    } else {
        l2.next = mergeTwoLists(l1, l2.next);
        return l2;
    }
}
```

Use Case

Combining sorted data from different sources such as merging user timelines, logs, or queues efficiently.

Interview Q&A

Q: How does the recursive merging work internally?

A: It compares the current nodes of both lists and recursively merges the rest, always choosing the smaller node as the next part of the merged list.

Q: What is the time complexity?

A: O(n + m), where n and m are the lengths of the two lists.

Topic: Reverse a Linked List (Iterative)

Definition:

Reversing the order of nodes in a singly linked list using a loop.

Code Example:

```
function reverseList(head) {
    let prev = null;
    let curr = head;

    while (curr) {
        let next = curr.next;
        curr.next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}
```

Use Case

Used in undo functionality, stack implementation, or solving problems that require processing in reverse.

Interview Q&A

Q: What's the key idea behind iterative reversal?

A: Rewire the next pointers of each node to point to the previous node.

Q: Time and space complexity?

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

Topic: Reverse a Linked List (Recursive)

Definition:

Reverses a linked list using recursive function calls and backtracking.

Code (Example):

```
function reverseList(head) {
    if (!head || !head.next) return head;

    let newHead = reverseList(head.next);
    head.next.next = head;
    head.next = null;

    return newHead;
}
```

Use Case

Elegant for problems where recursive backtracking naturally fits the logic like tree post-order traversal or stack unrolling.

Interview Q&A

Q: What is the base case in recursion?

A: When the node is null or there's only one node (! head.next), return that node.

Q: Any limitation of recursive approach?

A: Stack overflow risk on very large lists due to call stack depth.

Class 2: Detecting and Cleaning Up Linked Lists

Topic: Detect Cycle in Linked List (Floyd's Algorithm)

Definition:

Detects if a linked list contains a cycle using two pointers moving at different speeds.

Code (Example):

```
function hasCycle(head) {
    let slow = head;
    let fast = head;

    while (fast && fast.next) {
        slow = slow.next;
        fast = fast.next.next;

        if (slow === fast) return true;
    }
    return false;
}
```

Use Case

Crucial in identifying infinite loops in linked list operations or problems involving circular references.

Interview Q&A

Q: Why does the fast pointer eventually meet the slow one in a cycle?

A: Because fast moves 2 steps and slow 1 step; like a runner lapping another in a circular track.

Q: Time and space complexity?

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

Topic: Remove Duplicates from Sorted Linked List

Definition

Removes consecutive duplicate nodes from a sorted linked list.

Code (Example):

```
function deleteDuplicates(head) {
    let current = head;

    while (current && current.next) {
        if (current.val === current.next.val) {
            current.next = current.next.next;
        } else {
            current = current.next;
        }
    }
    return head;
}
```

Use Case:

Used in data cleanup tasks where duplicate records are not needed (e.g., de-duplicating user input, removing redundancy in logs).

🔷 Interview Q&A

Q: Why does this work only for sorted lists?

A: Because duplicates in unsorted lists may not be consecutive — this method relies on adjacent nodes being equal.

Q: Can this be done recursively?

A: Yes, by checking the current node and calling recursively for head.next.