Linked List Problems

Detect Loop in a Linked List: Implement an algorithm to detect if a given linked list contains a loop. This should be different from your existing cycle detection method.

Find Middle Element: Write a method to find the middle element of a linked list. Modify your existing code to handle edge cases.

Remove Duplicates from Unsorted List: Given an unsorted linked list, remove all duplicate nodes.

Intersection of Two Linked Lists: Given two linked lists, determine if they intersect and if they do, return the node where they intersect.

Flatten a Multilevel Doubly Linked List: Given a doubly linked list where nodes have a child pointer to another doubly linked list, flatten the list so that all nodes appear in a single-level doubly linked list.

Add Two Numbers Represented by Linked Lists: Represent two numbers as linked lists where each node contains a single digit and add the two numbers. The result should also be a linked list.

Sort a Linked List: Implement merge sort on a linked list. This should be different from your existing merge sort implementation.

Rotate a Linked List: Given a linked list and an integer k, rotate the linked list to the right by k places.

Remove Duplicates from Sorted List: Given a sorted linked list, remove all duplicate nodes.

Reverse a Doubly Linked List: Implement a function to reverse a doubly linked list, using both iterative and recursive methods.

Delete Nodes with Greater Value on Right: Given a linked list, remove all nodes which have a greater value on their right.

Merge Two Sorted Linked Lists: Given two sorted linked lists, merge them into a single sorted linked list.

Detect a Palindrome in a Linked List: Check if the linked list is a palindrome using a different approach from the one you’ve implemented.

Find the Starting Node of the Loop: Given a linked list that may contain a loop, find the node where the loop starts.

Find the Length of Loop: Given a linked list that contains a loop, find the length of the loop.

Remove Every Kth Node: Given a linked list and an integer k, remove every kth node from the linked list.

Reverse Nodes in K-Group: Given a linked list, reverse the nodes of the linked list in k-group and return its modified list.

Rotate Sublist: Given a linked list and two integers left and right, rotate the sublist of nodes from position left to right in the list.

Swap Nodes in Pairs: Given a linked list, swap every two adjacent nodes and return its head.

Copy List with Random Pointers: Given a linked list where each node has an additional random pointer which could point to any node in the list or null, make a deep copy of the linked list.

explain me problem statement with example in simple way and give java simple and optimizied code with time complexity

Let's walk through each of the linked list problems one by one, with simplified problem statements, example explanations, and optimized Java implementations.

---

1. Detect Loop in a Linked List

Problem Statement: Given a linked list, determine if it contains a cycle (loop) where a node's `next` pointer points back to a previous node.
Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5 -> 2 (loop back to node 2)`
- Output: `true`

Solution Approach:

- Use **Floyd's Cycle Detection Algorithm (Tortoise and Hare)**. One pointer moves one step at a time, and the other moves two steps. If they meet, there's a loop.

Java Code:

```java
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) { this.val = val; }
}

public class LinkedListProblems {

    // Detect loop using Floyd's Cycle detection
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) return false;
        ListNode slow = head, fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;      // Move slow by 1 step
            fast = fast.next.next; // Move fast by 2 steps
            if (slow == fast) return true; // Cycle detected
        }
        return false; // No cycle found
    }
}
```

Time Complexity: `O(n)` where `n` is the number of nodes.

---

2. Find Middle Element

Problem Statement: Find the middle element of a linked list. If there are two middle elements, return the second one.
Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5`
- Output: `3`

Solution Approach:

- Use **two-pointer technique**. Move one pointer at normal speed and the other at double speed. When the fast pointer reaches the end, the slow pointer will be at the middle.

Java Code:

```java
public ListNode findMiddle(ListNode head) {
    if (head == null) return null;

    ListNode slow = head, fast = head;

    while (fast != null && fast.next != null) {
        slow = slow.next;      // Move slow by 1 step
        fast = fast.next.next; // Move fast by 2 steps
    }

    return slow; // Slow pointer is now at the middle
}
```

Time Complexity: `O(n)`

---

3. Remove Duplicates from Unsorted List

Problem Statement: Given an unsorted linked list, remove all duplicate elements.
Example:

- Input: `1 -> 3 -> 2 -> 3 -> 1`
- Output: `1 -> 3 -> 2`

Solution Approach:

- Use a **HashSet** to track elements and remove duplicates.

Java Code:

```java
import java.util.HashSet;

public ListNode removeDuplicates(ListNode head) {
    if (head == null) return null;

    HashSet<Integer> set = new HashSet<>();
    ListNode current = head;
    ListNode prev = null;

    while (current != null) {
        if (set.contains(current.val)) {
            prev.next = current.next; // Skip duplicate
        } else {
            set.add(current.val);
            prev = current;
        }
        current = current.next;
    }

    return head;
}
```

Time Complexity: `O(n)`, Space Complexity: `O(n)` for storing elements in HashSet.

---

4. Intersection of Two Linked Lists

Problem Statement: Find the node where two linked lists intersect.
Example:

- Input:
    - List A: `1 -> 9 -> 1 -> 2 -> 4`
    - List B: `3 -> 2 -> 4`
    - Output: Node with value `2`

Solution Approach:

- Use two pointers, one for each list. Move both pointers forward, and when one reaches the end, move it to the start of the other list. They will eventually meet at the intersection point or reach null.

Java Code:

```java
public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
    if (headA == null || headB == null) return null;

    ListNode a = headA, b = headB;

    while (a != b) {
        a = (a == null) ? headB : a.next;
        b = (b == null) ? headA : b.next;
    }

    return a;
}
```

Time Complexity: `O(m + n)` where `m` and `n` are the lengths of the two linked lists.

---

5. Flatten a Multilevel Doubly Linked List

Problem Statement: Given a multilevel doubly linked list where nodes may have a child pointer to another doubly linked list, flatten it into a single-level list.
Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5 -> 6` (where node `3` has a child `7 -> 8`)
- Output: `1 -> 2 -> 3 -> 7 -> 8 -> 4 -> 5 -> 6`

Solution Approach:

- Use a stack to store nodes and process them level by level.

Java Code:

```java
class DoublyListNode {
    int val;
    DoublyListNode next, prev, child;
    DoublyListNode(int val) { this.val = val; }
}

public DoublyListNode flatten(DoublyListNode head) {
    if (head == null) return null;

    DoublyListNode current = head;
    Stack<DoublyListNode> stack = new Stack<>();

    while (current != null) {
        if (current.child != null) {
            if (current.next != null) {
                stack.push(current.next); // Save the next node to stack
            }
            current.next = current.child; // Flatten child level
            current.next.prev = current;
            current.child = null;
        }
        if (current.next == null && !stack.isEmpty()) {
            current.next = stack.pop();
            current.next.prev = current;
        }
        current = current.next;
    }

    return head;
}
```

Time Complexity: `O(n)` where `n` is the total number of nodes across all levels.

These are the first five problems. I will continue with the remaining problems in the next message!

give remaining

Let's continue with the remaining linked list problems.

---

6. Add Two Numbers Represented by Linked Lists

Problem Statement: Two numbers are represented by linked lists where each node contains a single digit. Add the two numbers and return the result as a linked list.
Example:

- Input:
    - List 1: `2 -> 4 -> 3` (represents 342)
    - List 2: `5 -> 6 -> 4` (represents 465)
- Output: `7 -> 0 -> 8` (represents 807)

Solution Approach:

- Traverse both lists and add corresponding digits. Handle the carry from one position to the next.

Java Code:

```java
public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode p = l1, q = l2, current = dummy;
    int carry = 0;

    while (p != null || q != null) {
        int x = (p != null) ? p.val : 0;
        int y = (q != null) ? q.val : 0;
        int sum = carry + x + y;
        carry = sum / 10;
        current.next = new ListNode(sum % 10);
        current = current.next;
        if (p != null) p = p.next;
        if (q != null) q = q.next;
    }

    if (carry > 0) {
        current.next = new ListNode(carry);
    }

    return dummy.next;
}
```

Time Complexity: `O(max(m, n))`, where `m` and `n` are the lengths of the two lists.

---

7. Sort a Linked List

Problem Statement: Sort a linked list using merge sort.
Example:

- Input: `4 -> 2 -> 1 -> 3`
- Output: `1 -> 2 -> 3 -> 4`

Solution Approach:

- Use **merge sort** to recursively split and merge the list.

Java Code:

```java
public ListNode mergeSort(ListNode head) {
    if (head == null || head.next == null) return head;

    // Split the list into two halves
    ListNode mid = getMiddle(head);
    ListNode nextOfMid = mid.next;
    mid.next = null;

    // Recursively sort the two halves
    ListNode left = mergeSort(head);
    ListNode right = mergeSort(nextOfMid);

    // Merge the sorted halves
    return merge(left, right);
}

private ListNode getMiddle(ListNode head) {
    if (head == null) return head;
    ListNode slow = head, fast = head.next;

    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }
    return slow;
}

private ListNode merge(ListNode left, ListNode right) {
    ListNode result = null;

    if (left == null) return right;
    if (right == null) return left;

    if (left.val <= right.val) {
        result = left;
        result.next = merge(left.next, right);
    } else {
        result = right;
        result.next = merge(left, right.next);
    }

    return result;
}
```

Time Complexity: `O(n log n)`.

---

8. Rotate a Linked List

Problem Statement: Rotate the list to the right by `k` places.
Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5`, `k = 2`
- Output: `4 -> 5 -> 1 -> 2 -> 3`

Solution Approach:

- Compute the length, find the rotation point, and rearrange the list.

Java Code:

```java
public ListNode rotateRight(ListNode head, int k) {
    if (head == null || k == 0) return head;

    ListNode current = head;
    int length = 1;

    // Find the length of the list
    while (current.next != null) {
        current = current.next;
        length++;
    }

    // Make the list circular
    current.next = head;

    // Find the new head after rotation
    k = k % length;
    int stepsToNewHead = length - k;
    ListNode newTail = head;

    for (int i = 1; i < stepsToNewHead; i++) {
        newTail = newTail.next;
    }

    ListNode newHead = newTail.next;
    newTail.next = null;

    return newHead;
}
```

Time Complexity: `O(n)`.

---

9. Remove Duplicates from Sorted List

Problem Statement: Given a sorted linked list, remove all duplicate elements.
Example:

- Input: `1 -> 1 -> 2 -> 3 -> 3`
- Output: `1 -> 2 -> 3`

Solution Approach:

- Traverse the list and skip over duplicate nodes.

Java Code:

```java
public ListNode removeDuplicatesSorted(ListNode head) {
    if (head == null) return null;

    ListNode current = head;
    while (current != null && current.next != null) {
        if (current.val == current.next.val) {
            current.next = current.next.next; // Skip duplicate
        } else {
            current = current.next;
        }
    }

    return head;
}
```

Time Complexity: `` `O(n)` ``.

---

10. Reverse a Doubly Linked List

Problem Statement: Reverse a doubly linked list.
Example:

- Input: `` `1 <-> 2 <-> 3` ``
- Output: `` `3 <-> 2 <-> 1` ``

Solution Approach:

- Swap the `` `next` `` and `` `prev` `` pointers for each node.

Java Code (Iterative):

```java
    public DoublyListNode reverseDoublyLinkedList(DoublyListNode head) {
        if (head == null) return null;

        DoublyListNode current = head;
        DoublyListNode prev = null;

        while (current != null) {
            DoublyListNode next = current.next;
            current.next = prev;
            current.prev = next;
            prev = current;
            current = next;
        }

        return prev; // New head of the reversed list
    }
```

Time Complexity: `` `O(n)` ``.

---

11. Delete Nodes with Greater Value on Right

Problem Statement: Remove all nodes from the linked list that have a greater value on their right.
Example:

- Input: `` `12 -> 15 -> 10 -> 11 -> 5 -> 6 -> 2 -> 3` ``
- Output: `` `15 -> 11 -> 6 -> 3` ``

Solution Approach:

- Reverse the list, delete nodes, and reverse it back.

Java Code:

```java
public ListNode deleteNodesWithGreaterRight(ListNode head) {
    head = reverseList(head);

    ListNode maxNode = head;
    ListNode current = head;

    while (current != null && current.next != null) {
        if (current.next.val < maxNode.val) {
            current.next = current.next.next;
        } else {
            current = current.next;
            maxNode = current;
        }
    }

    return reverseList(head);
}

private ListNode reverseList(ListNode head) {
    ListNode prev = null, current = head;
    while (current != null) {
        ListNode next = current.next;
        current.next = prev;
        prev = current;
        current = next;
    }
    return prev;
}
```

Time Complexity: `` `O(n)` ``.

---

12. Merge Two Sorted Linked Lists

Problem Statement: Merge two sorted linked lists into one sorted linked list.
Example:

- Input:
  - List 1: `` `1 -> 2 -> 4` ``
  - List 2: `` `1 -> 3 -> 4` ``
- Output: `` `1 -> 1 -> 2 -> 3 -> 4 -> 4` ``

Solution Approach:

- Merge the lists by comparing nodes.

Java Code:

```java
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    ListNode dummy = new ListNode(0);
    ListNode current = dummy;

    while (l1 != null && l2 != null) {
        if (l1.val <= l2.val) {
            current.next = l1;
            l1 = l1.next;
        } else {
            current.next = l2;
            l2 = l2.next;
        }
        current = current.next;
    }

    if (l1 != null) {
        current.next = l1;
    } else {
        current.next = l2;
    }

    return dummy.next;
}
```

Time Complexity: `` `O(m + n)` ``.

The remaining problems can be provided in the next message! Let me know if you'd like more!

give remaining

Here are the solutions to the remaining linked list problems:

---

13. Detect a Palindrome in a Linked List

Problem Statement: Check if the linked list is a palindrome (reads the same forward and backward).
Example:

- Input: `` `1 -> 2 -> 2 -> 1` ``
- Output: `` `true` ``

Solution Approach:

- Find the middle of the list, reverse the second half, and compare both halves.

Java Code:

```java
public boolean isPalindrome(ListNode head) {
    if (head == null || head.next == null) return true;

    ListNode slow = head, fast = head;

    // Find the middle of the list
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
    }

    // Reverse the second half of the list
    ListNode secondHalf = reverseList(slow);
    ListNode firstHalf = head;

    // Compare both halves
    while (secondHalf != null) {
        if (firstHalf.val != secondHalf.val) return false;
```

```java
            firstHalf = firstHalf.next;
            secondHalf = secondHalf.next;
        }

        return true;
    }

    private ListNode reverseList(ListNode head) {
        ListNode prev = null;
        while (head != null) {
            ListNode next = head.next;
            head.next = prev;
            prev = head;
            head = next;
        }
        return prev;
    }
```

Time Complexity: `` `O(n)` ``.

---

14. Find the Starting Node of the Loop

Problem Statement: Given a linked list with a loop, find the node where the loop starts.
Example:

- Input: `` `1 -> 2 -> 3 -> 4 -> 5 -> 2 (loop starts at node 2)` ``
- Output: Node with value `` `2` ``

Solution Approach:

- Use **Floyd's Cycle Detection Algorithm** to detect the loop, then find the starting node by resetting one pointer to the head and moving both pointers one step at a time until they meet.

Java Code:

```java
java

    public ListNode detectCycle(ListNode head) {
        if (head == null || head.next == null) return null;

        ListNode slow = head, fast = head;

        // Detect cycle using slow and fast pointers
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
            if (slow == fast) {
                break; // Cycle detected
            }
        }

        if (fast == null || fast.next == null) return null; // No cycle

        // Find the start of the loop
        slow = head;
        while (slow != fast) {
            slow = slow.next;
            fast = fast.next;
        }

        return slow; // Starting node of the loop
    }
```

Time Complexity: `O(n)`.

---

15. Find the Length of Loop

Problem Statement: Find the length of the loop in a linked list.
Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5 -> 2 (loop starts at node 2)`
- Output: `4`

Solution Approach:

- Use **Floyd's Cycle Detection Algorithm** to detect the loop. Once the loop is detected, start counting nodes from the meeting point until you come back to the same node.

Java Code:

```java
public int lengthOfLoop(ListNode head) {
    if (head == null || head.next == null) return 0;

    ListNode slow = head, fast = head;

    // Detect cycle
    while (fast != null && fast.next != null) {
        slow = slow.next;
        fast = fast.next.next;
        if (slow == fast) {
            return countLoopLength(slow); // Start counting the loop length
        }
    }

    return 0; // No loop
}

private int countLoopLength(ListNode node) {
    ListNode temp = node;
    int length = 1;

    while (temp.next != node) {
        temp = temp.next;
        length++;
    }

    return length;
}
```

Time Complexity: `O(n)`.

---

16. Remove Every Kth Node

Problem Statement: Remove every kth node from the linked list.
Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5 -> 6`, `k = 2`
- Output: `1 -> 3 -> 5`

Solution Approach:

- Traverse the list, counting the nodes, and remove every `kth` node.

Java Code:

```java
public ListNode removeEveryKthNode(ListNode head, int k) {
    if (head == null || k <= 1) return null;

    ListNode current = head;
    ListNode prev = null;
    int count = 1;

    while (current != null) {
        if (count % k == 0) {
            prev.next = current.next; // Remove kth node
        } else {
            prev = current;
        }
        current = current.next;
        count++;
    }

    return head;
}
```

Time Complexity: `O(n)`.

---

17. Reverse Nodes in K-Group

Problem Statement: Reverse the nodes of a linked list in groups of size `k`.
Example:

- Input: `1 -> 2 -> 3 -> 4 -> 5`, `k = 2`
- Output: `2 -> 1 -> 4 -> 3 -> 5`

Solution Approach:

- Reverse every `k` nodes and leave the remaining nodes if fewer than `k`.

Java Code:

```java
public ListNode reverseKGroup(ListNode head, int k) {
    if (head == null || k == 1) return head;

    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode prev = dummy, current = dummy, next = dummy;

    int length = 0;
    while (current.next != null) {
        current = current.next;
        length++;
    }

    while (length >= k) {
        current = prev.next;
        next = current.next;
        for (int i = 1; i < k; i++) {
            current.next = next.next;
```

```
                next.next = prev.next;
                prev.next = next;
                next = current.next;
            }
            prev = current;
            length -= k;
        }

        return dummy.next;
    }
```

Time Complexity: `` `O(n)` ``.

---

18. Rotate Sublist

Problem Statement: Rotate a sublist from position `` `left` `` to `` `right` `` in a linked list.
Example:

- Input: `` `1 -> 2 -> 3 -> 4 -> 5` ``, `` `left = 2` ``, `` `right = 4` ``
- Output: `` `1 -> 4 -> 3 -> 2 -> 5` ``

Solution Approach:

- Reverse the sublist from position `` `left` `` to `` `right` ``.

Java Code:

```java
    public ListNode rotateSublist(ListNode head, int left, int right) {
        if (head == null || left == right) return head;

        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode prev = dummy;

        // Move prev to the node before the left position
        for (int i = 1; i < left; i++) {
            prev = prev.next;
        }

        ListNode current = prev.next;
        ListNode next;

        // Reverse the sublist
        for (int i = 0; i < right - left; i++) {
            next = current.next;
            current.next = next.next;
            next.next = prev.next;
            prev.next = next;
        }

        return dummy.next;
    }
```

Time Complexity: `` `O(n)` ``.

---

19. Swap Nodes in Pairs

Problem Statement: Swap every two adjacent nodes and return the modified list.

Example:

- Input: `1 -> 2 -> 3 -> 4`
- Output: `2 -> 1 -> 4 -> 3`

Solution Approach:

- Swap nodes in pairs as you traverse the list.

Java Code:

```java
public ListNode swapPairs(ListNode head) {
    if (head == null || head.next == null) return head;

    ListNode dummy = new ListNode(0);
    dummy.next = head;
    ListNode prev = dummy;

    while (prev.next != null && prev.next.next != null) {
        ListNode first = prev.next;
        ListNode second = prev.next.next;

        first.next = second.next;
        second.next = first;
        prev.next = second;

        prev = first;
    }

    return dummy.next;
}
```

Time Complexity: `O(n)`.

---

20. Copy List with Random Pointers

Problem Statement: Given a linked list where each node has a `next` pointer and an additional `random` pointer that could point to any node in the list, clone the list.
Example:

- Input: `1 -> 2 -> 3`, where `random` pointers point to other nodes
- Output: A deep copy of the list.

Solution Approach:

- Create a deep copy by first interleaving the original nodes with their copies, then fixing the `next` and `random` pointers.

Java Code:

```java
class RandomListNode {
    int val;
    RandomListNode next, random;
    RandomListNode(int x) { this.val = x; }
}

public RandomListNode copyRandomList(RandomListNode head) {
    if (head == null) return null;

    RandomListNode current = head;

    // Step 1: Create new nodes interleaved with the original nodes
    while (current != null) {
        RandomListNode newNode = new RandomListNode(current.val);
        newNode.next = current.next;
        current.next = newNode;
        current = newNode.next;
    }

    // Step 2: Assign random pointers to the copied nodes
    current = head;
    while (current != null) {
        if (current.random != null) {
            current.next.random = current.random.next;
        }
        current = current.next.next;
    }

    // Step 3: Separate the original and copied lists
    current = head;
    RandomListNode copyHead = head.next;
    RandomListNode copy = copyHead;

    while (current != null) {
        current.next = current.next.next;
        if (copy.next != null) {
            copy.next = copy.next.next;
        }
        current = current.next;
        copy = copy.next;
    }

    return copyHead;
}
```

Time Complexity: `` `O(n)` ``.

These are the remaining linked list problems and their solutions. Let me know if you'd like further details or explanations!

ChatGPT can make mistakes. Check important info.