1. Reverse a Linked List

Detailed Problem

Given the head of a singly linked list, reverse the list and return the reversed list.

Input and Output

```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Logic Explanation

We will use three pointers: `prev`, `current`, and `next`. The `current` pointer starts at the head of the list. In each iteration, we reverse the link by pointing `current.next` to `prev`, then move `prev` and `current` one step forward.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseList(head: ListNode) -> ListNode:
    prev = None  # Previous pointer
    current = head  # Current pointer

    while current:
        next = current.next  # Store next node
        current.next = prev  # Reverse the link
        prev = current  # Move prev one step forward
        current = next  # Move current one step forward

    return prev  # New head of the reversed list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are using only a fixed amount of extra space.

---

2. Detect a Cycle in a Linked List

Detailed Problem

Given a linked list, determine if it has a cycle in it.

Input and Output

```
Input: head = [3,2,0,-4], pos = 1
Output: true (Cycle starts at node with value 2)
```

Logic Explanation

We can use Floyd's Tortoise and Hare algorithm. We have two pointers, `slow` and `fast`. The `slow` pointer moves one step at a time, while the `fast` pointer moves two steps. If there is a cycle, the fast pointer will eventually meet the slow pointer.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def hasCycle(head: ListNode) -> bool:
    slow = head  # Slow pointer
    fast = head  # Fast pointer

    while fast and fast.next:  # Check for the end of the list
        slow = slow.next  # Move slow by one
        fast = fast.next.next  # Move fast by two
        if slow == fast:  # Cycle detected
            return True

    return False  # No cycle detected
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), as we are using only a constant amount of extra space.

---

3. Find the Middle of a Linked List

Detailed Problem

Given the head of a linked list, return the middle node of the linked list. If there are two middle nodes, return the second middle node.

Input and Output

```
Input: head = [1,2,3,4,5]
Output: [3,4,5]  (Middle node is 3)
```

Logic Explanation

We can again use the two-pointer technique. The `slow` pointer moves one step at a time, while the `fast` pointer moves two steps. When the `fast` pointer reaches the end, the `slow` pointer will be at the middle.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def middleNode(head: ListNode) -> ListNode:
    slow = head  # Slow pointer
    fast = head  # Fast pointer

    while fast and fast.next:  # Check for the end of the list
        slow = slow.next  # Move slow by one
        fast = fast.next.next  # Move fast by two

    return slow  # Middle node
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we're using only a fixed amount of extra space.

---

4. Merge Two Sorted Linked Lists

Detailed Problem

Merge two sorted linked lists and return it as a new sorted list. The new list should be made by splicing together the nodes of the first two lists.

Input and Output

```
Input: l1 = [1,2,4], l2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Logic Explanation

We can create a dummy node to help with the merging process. We'll compare the values of the nodes from both lists and append the smaller node to the merged list until we reach the end of one of the lists. Finally, we'll attach the remaining nodes from the other list.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode(0)  # Dummy node to start the merged list
    tail = dummy  # Pointer to construct the merged list

    while l1 and l2:  # While both lists are non-empty
        if l1.val < l2.val:
            tail.next = l1  # Append l1 node
            l1 = l1.next  # Move l1 pointer
        else:
            tail.next = l2  # Append l2 node
            l2 = l2.next  # Move l2 pointer
        tail = tail.next  # Move tail pointer

    tail.next = l1 if l1 else l2  # Append remaining nodes
    return dummy.next  # Return merged list
```

Time and Space Complexity

- **Time Complexity**: O(n + m), where n and m are the lengths of the two lists. We traverse both lists entirely.

- **Space Complexity**: O(1), as we are merging in place without using extra space.

---

5. Remove Nth Node From End of List

Detailed Problem

Given the head of a linked list, remove the nth node from the end of the list and return its head.

Input and Output

```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]  (Node with value 4 is removed)
```

Logic Explanation

We can use the two-pointer technique again. First, we move the `fast` pointer n steps ahead. Then, we move both `fast` and `slow` pointers until `fast` reaches the end. The `slow` pointer will be right before the node we want to remove.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def removeNthFromEnd(head: ListNode, n: int) -> ListNode:
    dummy = ListNode(0, head)  # Create a dummy node
    fast = slow = dummy  # Initialize two pointers

    for _ in range(n):  # Move fast n steps ahead
        fast = fast.next

    while fast.next:  # Move both pointers until fast reaches the end
        fast = fast.next
        slow = slow.next

    slow.next = slow.next.next  # Remove the nth node
    return dummy.next  # Return the modified list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), as we are using a constant amount of extra space.

---

6. Linked List Cycle II (Start of Cycle)

Detailed Problem

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Input and Output

```
Input: head = [3,2,0,-4], pos = 1
Output: Node with value 2 (start of cycle)
```

Logic Explanation

Using Floyd’s algorithm, we first detect the cycle. Then, to find the entry point, we keep one pointer at the head and move both pointers one step at a time. The point at which they meet is the start of the cycle.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def detectCycle(head: ListNode) -> ListNode:
    slow = fast = head  # Initialize slow and fast pointers

    # Detect cycle using fast and slow pointers
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:  # Cycle detected
            # Find entry point of the cycle
            entry = head
            while entry != slow:
                entry = entry.next
                slow = slow.next
            return entry  # Entry point of the cycle

    return None  # No cycle detected
```

Time and Space Complexity

- **Time Complexity**: O(n), where n

  is the number of nodes in the linked list.

- **Space Complexity**: O(1), as we are using a constant amount of extra space.

---

7. Palindrome Linked List

Detailed Problem

Given the head of a singly linked list, determine if it is a palindrome.

Input and Output

```
Input: head = [1,2,2,1]
Output: true
```

Logic Explanation

We can use the fast and slow pointer technique to find the middle of the linked list, then reverse the second half of the list and compare it with the first half.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def isPalindrome(head: ListNode) -> bool:
    if not head or not head.next:
        return True

    slow = fast = head  # Initialize slow and fast pointers

    # Find the middle of the linked list
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Reverse the second half
    prev = None
    while slow:
        next_node = slow.next
        slow.next = prev
        prev = slow
        slow = next_node

    # Compare the two halves
    left, right = head, prev
    while right:  # Only need to compare till the end of the reversed half
        if left.val != right.val:
            return False
        left = left.next
        right = right.next

    return True
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we only use a fixed amount of extra space.

---

8. Intersection of Two Linked Lists

Detailed Problem

Given the heads of two singly linked lists, determine if the two lists intersect and return the intersecting node. If they do not intersect, return null.

Input and Output

```
Input:
A: [4,1,8,4,5]
B: [5,0,1,8,4,5]
Output: Node with value 8 (the intersection node)
```

Logic Explanation

We can find the lengths of both lists and calculate the difference. Then we advance the longer list by the difference in lengths. After that, we can traverse both lists together until we find the intersection.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def getIntersectionNode(headA: ListNode, headB: ListNode) -> ListNode:
    if not headA or not headB:
        return None

    # Get the lengths of both lists
    def getLength(node):
        length = 0
        while node:
            length += 1
            node = node.next
        return length

    lenA = getLength(headA)
    lenB = getLength(headB)

    # Adjust the longer list
    while lenA > lenB:
        headA = headA.next
        lenA -= 1
    while lenB > lenA:
        headB = headB.next
        lenB -= 1

    # Traverse both lists until we find the intersection
    while headA and headB:
        if headA == headB:
            return headA
        headA = headA.next
        headB = headB.next

    return None
```

Time and Space Complexity

- **Time Complexity**: O(n + m), where n and m are the lengths of the two lists.
- **Space Complexity**: O(1), since we are using a constant amount of extra space.

---

9. Remove Duplicates from Sorted List

Detailed Problem

Given a sorted linked list, delete all duplicates such that each element appears only once.

Input and Output

```
Input: head = [1,1,2]
Output: [1,2]
```

Logic Explanation

We can use a single pointer to traverse the list. If the current node's value is the same as the next node's value, we skip the next node.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def deleteDuplicates(head: ListNode) -> ListNode:
    current = head  # Pointer to traverse the list

    while current and current.next:
        if current.val == current.next.val:  # Check for duplicates
            current.next = current.next.next  # Skip duplicate node
        else:
            current = current.next  # Move to the next node

    return head  # Return the modified list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), as we are modifying the list in place.

---

10. Remove Duplicates from Unsorted List

Detailed Problem

Given an unsorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

Input and Output

```
Input: head = [4, 3, 2, 7, 8, 3, 2]
Output: [4, 7, 8]
```

Logic Explanation

We can use a hash set to track the occurrences of each value. After that, we traverse the list again to construct a new list that only includes values that appeared once.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def deleteDuplicates(head: ListNode) -> ListNode:
    if not head:
        return head

    count = {}  # Dictionary to count occurrences
    current = head

    # First pass to count occurrences
    while current:
        count[current.val] = count.get(current.val, 0) + 1
        current = current.next

    # Create a dummy node for the new list
    dummy = ListNode(0)
    new_tail = dummy
    current = head

    # Second pass to add only unique values
    while current:
        if count[current.val] == 1:
            new_tail.next = current  # Append to new list
            new_tail = new_tail.next
        current = current.next

    new_tail.next = None  # End of new list
    return dummy.next  # Return the new list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list (two passes).
- **Space Complexity**: O(n), due to the additional space used for the hash set.

---

11. Copy List with Random Pointer

Detailed Problem

Given a linked list where each node contains an additional random pointer which could point to any node in the list or null, return a deep copy of the list.

Input and Output

```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

Logic Explanation

We can use a hash map to maintain a mapping from the original nodes to their copies. First, we create a copy of each node and store it in the map. Then, we assign the next and random pointers for each copied node.

Code

```python
class RandomListNode:
    def __init__(self, val=0, next=None, random=None):
        self.val = val
        self.next = next
        self.random = random

def copyRandomList(head: RandomListNode) -> RandomListNode:
    if not head:
        return None

    # Step 1: Create copies of each node and store in the hash map
    mapping = {}
    current = head
    while current:
        mapping[current] = RandomListNode(current.val)
        current = current.next

    # Step 2: Assign next and random pointers for copied nodes
    current = head
    while current:
        copy_node = mapping[current]
        copy_node.next = mapping.get(current.next)  # Assign next
        copy_node.random = mapping.get(current.random)  # Assign random
        current = current.next

    return mapping[head]  # Return the head of the copied list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the list (two passes).
- **Space Complexity**: O(n), due to the hash map used for mapping nodes.

12. Reorder List

Detailed Problem

Given a linked list, reorder it such that the nodes follow a specific pattern: L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → ...

Input and Output

```
Input: head = [1,2,3,4]
Output: [1,4,2,3]
```

Logic Explanation

We can split the linked list into two halves, reverse the second half, and then merge the two halves in the specified order.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reorderList(head: ListNode) -> None:
    if not head or not head.next:
        return

    # Step 1: Find the middle of the list using slow and fast pointers
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Step 2: Reverse the second half of the list
    prev, current = None, slow
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node

    # Step 3: Merge the two halves
    first, second = head, prev
    while second.next:
        temp1, temp2 = first.next, second.next
        first.next = second
        second.next = temp1
        first, second = temp1, temp2
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list (three passes).
- **Space Complexity**: O(1), as we are rearranging the nodes in place.

---

13. Flatten a Multilevel Doubly Linked List

Detailed Problem

You are given a doubly linked list which may have one or more child pointers to a separate doubly linked list. Flatten the list so that all the nodes appear in a single-level, doubly linked list.

Input and Output

```
Input: head = [1,2,3,4,5,null,6,null,null,7,8]
Output: [1,2,3,7,8,4,5,6]
```

Logic Explanation

We can use a depth-first search (DFS) approach. For each node, we traverse through its children, flattening each child list recursively and linking them back into the main list.

Code

```python
class Node:
    def __init__(self, val=0, prev=None, next=None, child=None):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child

def flatten(head: Node) -> Node:
    if not head:
        return None

    # Helper function to recursively flatten the list
    def flatten_dfs(node):
        if not node:
            return node

        # Pointers to track the last node in the flattened list
        prev = None
        current = node

        while current:
            if current.child:
                # Flatten the child list
                child_tail = flatten_dfs(current.child)

                # Connect current node with the child
                current.next = current.child
                current.child.prev = current
                current.child = None  # Remove the child pointer

                if child_tail:
                    # Connect the end of the child list to the next node
                    child_tail.next = current.next
                    if current.next:
                        current.next.prev = child_tail

            prev = current
            current = current.next

        return prev  # Return the last node of the flattened list

    flatten_dfs(head)  # Start flattening
    return head  # Return the head of the flattened list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the list.
- **Space Complexity**: O(1), as we are modifying the list in place.

---

14. LRU Cache

Detailed Problem

Design a data structure that follows the constraints of a least recently used (LRU) cache. Implement the `LRUCache` class with methods to get and put values.

Input and Output

```
Input:
LRUCache cache = new LRUCache(2);
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
```

Logic Explanation

We can use a combination of a doubly linked list and a hash map. The linked list will maintain the order of usage, while the hash map will provide O(1) access to the cache values.

Code

```python
class DLinkedNode:
    def __init__(self, key=0, value=0):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}  # Hash map for key-value pairs
        self.head = DLinkedNode()  # Dummy head
        self.tail = DLinkedNode()  # Dummy tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node: DLinkedNode):
        node.prev.next = node.next
        node.next.prev = node.prev

    def _add(self, node: DLinkedNode):
        node.prev = self.tail.prev
        node.next = self.tail
        self.tail.prev.next = node
        self.tail.prev = node

    def get(self, key: int) -> int:
        if key in self.cache:
            node = self.cache[key]
            self._remove(node)
            self._add(node)
            return node.value
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self._remove(self.cache[key])
        node = DLinkedNode(key, value)
        self.cache[key] = node
        self._add(node)
        if len(self.cache) > self.capacity:
            # Remove the least recently used item
            lru = self.head.next
            self._remove(lru)
            del self.cache[lru.key]
```

Time and Space Complexity

- **Time Complexity**: O(1) for both `get` and `put` operations.
- **Space Complexity**: O(capacity), as we maintain a hash map and a linked list of at most `capacity` nodes.

---

15. Design a Linked List

Detailed Problem

Design a linked list that supports the following operations: `addAtHead`, `addAtTail`, `addAtIndex`, `deleteAtIndex`, and `get`.

Input and Output

```
MyLinkedList linkedList = new MyLinkedList();
linkedList.addAtHead(1);         // linked list becomes 1
linkedList.addAtTail(3);         // linked list becomes 1 -> 3
linkedList.addAtIndex(1, 2);     // linked list becomes 1 -> 2 -> 3
linkedList.get(1);                // returns 2
linkedList.deleteAtIndex(1);     // now the linked list is 1 -> 3
```

Logic Explanation

We will create a class that represents the linked list and implement the required methods. We will maintain a reference to the head of the list and dynamically create or

remove nodes as needed.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class MyLinkedList:
    def __init__(self):
        self.head = None
        self.size = 0

    def get(self, index: int) -> int:
        if index < 0 or index >= self.size:
            return -1

        current = self.head
        for _ in range(index):
            current = current.next
        return current.val

    def addAtHead(self, val: int) -> None:
        new_node = ListNode(val)
        new_node.next = self.head
        self.head = new_node
        self.size += 1

    def addAtTail(self, val: int) -> None:
        new_node = ListNode(val)
        if not self.head:
            self.head = new_node
        else:
            current = self.head
            while current.next:
                current = current.next
            current.next = new_node
        self.size += 1

    def addAtIndex(self, index: int, val: int) -> None:
        if index < 0 or index > self.size:
            return
        if index == 0:
            self.addAtHead(val)
        else:
            new_node = ListNode(val)
            current = self.head
            for _ in range(index - 1):
                current = current.next
            new_node.next = current.next
```

```
            current.next = new_node
            self.size += 1

    def deleteAtIndex(self, index: int) -> None:
        if index < 0 or index >= self.size:
            return

        if index == 0:
            self.head = self.head.next
        else:
            current = self.head
            for _ in range(index - 1):
                current = current.next
            current.next = current.next.next
        self.size -= 1
```

Time and Space Complexity

- **Time Complexity**: O(n) for `addAtIndex`, `deleteAtIndex`, and `get`; O(1) for `addAtHead` and `addAtTail`.
- **Space Complexity**: O(1) for operations, but O(n) for the space occupied by nodes.

---

16. Split Linked List in Parts

Detailed Problem

Given the head of a linked list and an integer k, split the linked list into k consecutive parts.

Input and Output

```
Input: head = [1,2,3], k = 5
Output: [[1],[2],[3],[],[]]
```

Logic Explanation

First, we determine the length of the linked list. Then we calculate the size of each part and the number of extra nodes. We traverse the list and split it into k parts accordingly.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def splitListToParts(head: ListNode, k: int):
    # First, calculate the length of the linked list
    length = 0
    current = head
    while current:
        length += 1
        current = current.next

    part_length = length // k  # Base size of each part
    extra_parts = length % k    # Extra nodes to distribute

    parts = []
    current = head
    for i in range(k):
        part_head = current
        for j in range(part_length + (1 if i < extra_parts else 0) - 1):
            if current:
                current = current.next
        if current:
            next_part = current.next
            current.next = None
            current = next_part
        parts.append(part_head)

    return parts
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(k), since we store the heads of the k parts in a list.

---

17. Convert Binary Search Tree to Sorted Doubly Linked List

Detailed Problem

Convert a binary search tree to a sorted doubly linked list in-place.

Input and Output

```
Input: root = [4,2,5,1,3]
Output: [1,2,3,4,5]
```

Logic Explanation

We can perform an in-order traversal of the binary search tree to create the doubly linked list. We maintain a previous pointer to link the nodes as we traverse.

Code

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def treeToDoublyList(root: TreeNode) -> TreeNode:
    if not root:
        return None

    def inorder(node):
        nonlocal prev, head
        if not node:
            return
        inorder(node.left)

        if prev:
            prev.right = node
            node.left = prev
        else:
            head = node
        prev = node

        inorder(node.right)

    prev = None
    head = None
    inorder(root)

    # Connect head and tail to make it circular
    if head and prev:
        head.left = prev
        prev.right = head

    return head
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the tree.
- **Space Complexity**: O(h), where h is the height of the tree due to the recursion stack.

---

18. Add Two Numbers (Linked List Representation)

Detailed Problem

Given two non-empty linked lists representing two non-negative integers, add the two numbers and return it as a linked list.

Input and Output

```
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]  (342 + 465 = 807)
```

Logic Explanation

We can iterate through both linked lists, adding the corresponding digits along with a carry from the previous addition. We create a new linked list to store the result.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def addTwoNumbers(l1: ListNode, l2: ListNode) -> ListNode:
    dummy_head = ListNode(0)  # Dummy node to simplify the result list
    current = dummy_head
    carry = 0

    while l1 or l2 or carry:
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0
        total = val1 + val2 + carry
        carry = total // 10  # Calculate carry for the next digit
        current.next = ListNode(total % 10)  # Create new node
        current = current.next  # Move to the next node

        if l1:
            l1 = l1.next  # Move to the next node in l1
        if l2:
            l2 = l2.next  # Move to the next node in l2

    return dummy_head.next  # Return the result list
```

Time and Space Complexity

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

- **Space Complexity**: O(max(n, m)), as we create a new linked list for the result.

---

19. Remove Linked List Elements

Detailed Problem

Remove all elements from a linked list that have a specific value.

Input and Output

```
Input: head = [1,2,6,3,4,5,6], val = 6
Output: [1,2,3,4,5]
```

Logic Explanation

We can iterate through the linked list while maintaining a previous pointer to help remove nodes that match the target value.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def removeElements(head: ListNode, val: int) -> ListNode:
    dummy = ListNode(0)  # Dummy node to handle edge cases
    dummy.next = head
    current = dummy

    while current and current.next:
        if current.next.val == val:
            current.next = current.next.next  # Skip the node
        else:
            current = current.next  # Move to the next node

    return dummy.next  # Return the modified list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

20. Odd Even Linked List

Detailed Problem

Given a linked list, reorder it so that all odd-indexed nodes appear before the even-indexed nodes.

Input and Output

```
Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]
```

Logic Explanation

We can use two pointers to maintain separate lists for odd and even indexed nodes, and then connect the end of the odd list to the head of the even list.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val


        self.next = next

def oddEvenList(head: ListNode) -> ListNode:
    if not head or not head.next:
        return head

    odd = head
    even = head.next
    even_head = even  # Keep the start of the even list

    while even and even.next:
        odd.next = even.next  # Link odd nodes
        odd = odd.next
        even.next = odd.next  # Link even nodes
        even = even.next

    odd.next = even_head  # Connect odd list to even list
    return head
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are rearranging the nodes in place.

---

21. Merge Two Sorted Lists

Detailed Problem

Merge two sorted linked lists into one sorted linked list.

Input and Output

```
Input: l1 = [1,2,4], l2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Logic Explanation

We can use a dummy node to simplify the merging process. We compare the heads of the two lists, appending the smaller value to the result list and moving the pointer of the respective list.

Code

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode(0)  # Dummy node to simplify the merging
    current = dummy

    while l1 and l2:
        if l1.val < l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    current.next = l1 if l1 else l2  # Append the remaining nodes
    return dummy.next  # Return the merged list
```

Time and Space Complexity

- **Time Complexity**: O(n + m), where n and m are the lengths of the two lists.
- **Space Complexity**: O(1), as we merge the lists in place.

---

22. Palindrome Linked List

Detailed Problem

Determine if a linked list is a palindrome.

Input and Output

```
Input: head = [1,2,2,1]
Output: true
```

Logic Explanation

We can use a two-pointer approach to find the middle of the list, then reverse the second half and compare it with the first half.

Code

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def isPalindrome(head: ListNode) -> bool:
    if not head or not head.next:
        return True

    # Step 1: Find the middle of the linked list
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Step 2: Reverse the second half of the list
    prev, current = None, slow
    while current:
        next_node = current.next
        current.next = prev
        prev = current
        current = next_node

    # Step 3: Compare the first and second halves
    first_half, second_half = head, prev
    while second_half:  # Only need to check the second half
        if first_half.val != second_half.val:
            return False
        first_half = first_half.next
        second_half = second_half.next

    return True
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we modify the list in place.

---

23. Find the Middle of the Linked List

Detailed Problem

Find the middle node of a linked list. If there are two middle nodes, return the second middle node.

Input and Output

```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
```

Logic Explanation

We can use the slow and fast pointer approach. The slow pointer moves one step at a time, while the fast pointer moves two steps. When the fast pointer reaches the end, the slow pointer will be at the middle.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def middleNode(head: ListNode) -> ListNode:
    slow, fast = head, head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    return slow  # Return the middle node
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we only use two pointers.

---

24. Intersection of Two Linked Lists

Detailed Problem

Determine the node at which two singly linked lists intersect.

Input and Output

```
Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5]
Output: [8,4,5]
```

Logic Explanation

We can use two pointers. The first pointer starts from the head of the first list and the second from the head of the second list. When they reach the end, we redirect them to the head of the other list. They will meet at the intersection point or both will reach the end simultaneously.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def getIntersectionNode(headA: ListNode, headB: ListNode) -> ListNode:
    if not headA or not headB:
        return None

    pointerA, pointerB = headA, headB

    while pointerA != pointerB:
        pointerA = pointerA.next if pointerA else headB
        pointerB = pointerB.next if pointerB else headA

    return pointerA  # Will be the intersection node or None
```

Time and Space Complexity

- **Time Complexity**: O(n + m), where n and m are the lengths of the two lists.
- **Space Complexity**: O(1), since we only use two pointers.

---

25. Remove Nth Node From End of List

Detailed Problem

Remove the n-th node from the end of the list and return its head.

Input and Output

```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

Logic Explanation

We can use a two-pointer technique. We move the first pointer n steps ahead. Then, we move both pointers until the first pointer reaches the end. The second pointer will point to the node before the one we need to remove.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def removeNthFromEnd(head: ListNode, n: int) -> ListNode:
    dummy = ListNode(0)  # Dummy node to handle edge cases
    dummy.next = head
    first = dummy
    second = dummy

    # Move first pointer n + 1 steps ahead
    for _ in range(n + 1):
        first = first.next

    # Move both pointers until the first pointer reaches the end
    while first:
        first = first.next
        second = second.next

    # Remove the n-th node
    second.next = second.next.next
    return dummy.next  # Return the modified list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

26. Reverse Nodes in k-Group

Detailed Problem

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

Input and Output

```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
```

Logic Explanation

We can use a dummy node to simplify the process. For every k nodes, we reverse the linked list and connect the reversed part back to the previous part.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseKGroup(head: ListNode, k: int) -> ListNode:
    dummy = ListNode(0)
    dummy.next = head
    prev_group_end = dummy

    while True:
        kth_node = prev_group_end
        for i in range(k):  # Check if there are k nodes left
            kth_node = kth_node.next
            if not kth_node:
                return dummy.next

        # Reverse k nodes
        group_start = prev_group_end.next
        prev, curr = None, group_start
        for _ in range(k):
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node

        # Connect reversed part with previous part and next part
        prev_group_end.next = prev
        group_start.next = curr
        prev_group_end = group_start
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

27. Rotate List

Detailed Problem

Given a linked list, rotate the list to the right by k places.

Input and Output

```
Input: head = [1,2,3

,4,5], k = 2
Output: [4,5,1,2,3]
```

Logic Explanation

We can first find the length of the list and then determine the effective number of rotations. After that, we can break the list at the appropriate point and reconnect it.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def rotateRight(head: ListNode, k: int) -> ListNode:
    if not head or not head.next or k == 0:
        return head

    # Step 1: Get the length of the list
    length = 1
    tail = head
    while tail.next:
        tail = tail.next
        length += 1

    # Step 2: Compute the effective number of rotations
    k %= length
    if k == 0:
        return head

    # Step 3: Find the new tail and new head
    new_tail = head
    for _ in range(length - k - 1):
        new_tail = new_tail.next

    new_head = new_tail.next
    new_tail.next = None
    tail.next = head  # Connect the end to the start

    return new_head
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

28. Copy List with Random Pointer

Detailed Problem

A linked list is given such that each node contains an additional random pointer, which could point to any node in the list or None. We need to return a deep copy of the list.

Input and Output

```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

Logic Explanation

We can use a hashmap to store the mapping between the original nodes and their copies. First, we copy all the nodes, and then we set the random pointers accordingly.

Code

```python
class Node:
    def __init__(self, val=0, next=None, random=None):
        self.val = val
        self.next = next
        self.random = random

def copyRandomList(head: Node) -> Node:
    if not head:
        return None

    # Step 1: Create a mapping of original nodes to their copies
    old_to_new = {}
    current = head
    while current:
        old_to_new[current] = Node(current.val)
        current = current.next

    # Step 2: Assign next and random pointers
    current = head
    while current:
        copy = old_to_new[current]
        copy.next = old_to_new.get(current.next)  # Assign next
        copy.random = old_to_new.get(current.random)  # Assign random
        current = current.next

    return old_to_new[head]  # Return the head of the copied list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(n), since we are storing the mapping in a hashmap.

---

29. Flatten a Multilevel Doubly Linked List

Detailed Problem

Flatten a multilevel doubly linked list into a single-level doubly linked list.

Input and Output

```
Input: head = 1 - 2 - 3 - 4 - 5 - 6 - 7
Output: 1 - 2 - 3 - 7 - 6 - 5 - 4
```

Logic Explanation

We can use a stack to traverse the multilevel list. For each node, we push its next node and then push the child node to the stack to continue flattening.

Code

```
class Node:
    def __init__(self, val=0, prev=None, next=None, child=None):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child

def flatten(head: Node) -> Node:
    if not head:
        return None

    dummy = Node(0)  # Dummy node to help with the flat structure
    stack = [head]
    prev = dummy

    while stack:
        current = stack.pop()
        prev.next = current
        current.prev = prev

        if current.next:
            stack.append(current.next)
        if current.child:
            stack.append(current.child)
            current.child = None  # Remove the child pointer

        prev = current

    return dummy.next  # Return the flattened list
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the multilevel list.
- **Space Complexity**: O(n), due to the stack used for traversal.

---

30. Linked List Cycle

Detailed Problem

Determine if a linked list has a cycle in it.

Input and Output

```
Input: head = [3,2,0,-4], pos = 1
Output: true
```

Logic Explanation

We can use the Floyd's Tortoise and Hare algorithm, where we use two pointers moving at different speeds. If there is a cycle, they will eventually meet.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def hasCycle(head: ListNode) -> bool:
    if not head:
        return False

    slow, fast = head, head

    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            return True

    return False
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we only use two pointers.

---

31. Linked List Cycle II

Detailed Problem

Given a linked list, return the node where the cycle begins. If there is no cycle, return `null`.

Input and Output

```
Input: head = [3,2,0,-4], pos = 1
Output: 2
```

Logic Explanation

Using Floyd's Tortoise and Hare algorithm, we can first detect if a cycle exists. If a cycle is found, we can then find the entry point by moving one pointer to the head and the other to the meeting point; they will meet at the cycle's starting node.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def detectCycle(head: ListNode) -> ListNode:
    if not head:
        return None

    slow, fast = head, head

    # Step 1: Detect cycle
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next
        if slow == fast:
            # Step 2: Find the entrance to the cycle
            entry = head
            while entry != slow:
                entry = entry.next
                slow = slow.next
            return entry

    return None
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we only use two pointers.

---

32. Reverse Linked List II

Detailed Problem

Reverse a linked list from position m to n. Do it in one-pass.

Input and Output

```
Input: head = [1,2,3,4,5], m = 2, n = 4
Output: [1,4,3,2,5]
```

Logic Explanation

We can first traverse the list to find the nodes before the m-th node and the n-th node. We then reverse the nodes between these positions and reconnect the list.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseBetween(head: ListNode, m: int, n: int) -> ListNode:
    if not head or m == n:
        return head

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

    # Step 1: Move `prev` to the node before position m
    for _ in range(m - 1):
        prev = prev.next

    # Step 2: Reverse the sublist from m to n
    reverse_start = prev.next
    reverse_end = reverse_start
    for _ in range(n - m):
        temp = reverse_end.next
        reverse_end.next = temp.next
        temp.next = prev.next
        prev.next = temp

    return dummy.next
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

33. Reorder List

Detailed Problem

Reorder the list to be in the following format: L0 → Ln → L1 → Ln-1 → L2 → Ln-2 → …

Input and Output

```
Input: head = [1,2,3,4]
Output: [1,4,2,3]
```

Logic Explanation

We can find the middle of the list, reverse the second half, and then merge the two halves.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reorderList(head: ListNode) -> None:
    if not head or not head.next:
        return

    # Step 1: Find the middle of the linked list
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Step 2: Reverse the second half of the list
    prev, curr = None, slow
    while curr:
        next_node = curr.next
        curr.next = prev
        prev = curr
        curr = next_node

    # Step 3: Merge the two halves
    first, second = head, prev
    while second.next:  # While there are nodes in the second half
        temp1, temp2 = first.next, second.next
        first.next = second
        second.next = temp1
        first, second = temp1, temp2
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

34. Kth Node From End of List

Detailed Problem

Find the k-th node from the end of a singly linked list.

Input and Output

```
Input: head = [1,2,3,4,5], k = 2
Output: 4
```

Logic Explanation

We can use a two-pointer approach. The first pointer advances k nodes ahead, then both pointers move together until the first pointer reaches the end.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def kthToLast(head: ListNode, k: int) -> ListNode:
    first = head
    for _ in range(k):
        first = first.next

    second = head
    while first:
        first = first.next
        second = second.next

    return second
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are using only a constant amount of space.

35. LRU Cache

Detailed Problem

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache. Implement the `get` and `put` methods.

Input and Output

```
Input:
LRUCache cache = new LRUCache(2); // capacity = 2
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);       // returns 1
cache.put(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
```

Logic Explanation

We can use a combination of a doubly linked list and a hashmap. The linked list maintains the order of access, while the hashmap provides O(1) access to cache items.

Code

```python
class Node:
    def __init__(self, key=0, val=0):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}  # key: Node
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def _add_to_front(self, node):
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node

    def get(self, key: int) -> int:
        if key in self.cache:
            node = self.cache[key]
            self._remove(node)
            self._add_to_front(node)
            return node.val
        return -1

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self._remove(self.cache[key])
        elif len(self.cache) >= self.capacity:
            lru = self.tail.prev
            self._remove(lru)
            del self.cache[lru.key]

        new_node = Node(key, value)
        self.cache[key] = new_node
        self._add_to_front(new_node)
```

Time and Space Complexity

- **Time Complexity**: O(1) for both `get` and `put`.
- **Space Complexity**: O(capacity), for storing the cache.

---

36. Merge K Sorted Lists

Detailed Problem

Merge k sorted linked lists and return it as one sorted list.

Input and Output

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
```

Logic Explanation

We can use a min-heap (priority queue) to keep track of the smallest elements from each list and build the merged list accordingly.

Code

```
import heapq

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeKLists(lists):
    min_heap = []

    # Step 1: Add the head of each list to the heap
    for i, head in enumerate(lists):
        if head:
            heapq.heappush(min_heap, (head.val, i, head

))

    dummy = ListNode(0)
    current = dummy

    # Step 2: Extract the smallest node and add the next node from the same list to the heap
    while min_heap:
        _, i, node = heapq.heappop(min_heap)
        current.next = node
        current = current.next
        if node.next:
            heapq.heappush(min_heap, (node.next.val, i, node.next))

    return dummy.next
```

Time and Space Complexity

- **Time Complexity**: O(n log k), where n is the total number of nodes and k is the number of lists.
- **Space Complexity**: O(k), for the heap.

---

37. Add Two Numbers

Detailed Problem

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order and each of their nodes contains a single digit. Add the two numbers and return it as a linked list.

Input and Output

```
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
```

Logic Explanation

We can iterate through both linked lists, adding corresponding digits along with any carry from the previous addition.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def addTwoNumbers(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode(0)
    current = dummy
    carry = 0

    while l1 or l2 or carry:
        sum_value = carry
        if l1:
            sum_value += l1.val
            l1 = l1.next
        if l2:
            sum_value += l2.val
            l2 = l2.next

        carry = sum_value // 10
        current.next = ListNode(sum_value % 10)
        current = current.next

    return dummy.next
```

Time and Space Complexity

- **Time Complexity**: O(max(n, m)), where n and m are the lengths of the two linked lists.
- **Space Complexity**: O(1), since we are using a constant amount of extra space.

---

38. Swap Nodes in Pairs

Detailed Problem

Given a linked list, swap every two adjacent nodes and return its head.

Input and Output

```
Input: head = [1,2,3,4]
Output: [2,1,4,3]
```

Logic Explanation

We can use a dummy node to simplify swapping pairs and maintain pointers to previous and current nodes.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def swapPairs(head: ListNode) -> ListNode:
    dummy = ListNode(0)
    dummy.next = head
    current = dummy

    while current.next and current.next.next:
        first = current.next
        second = current.next.next

        # Step 1: Swap the nodes
        first.next = second.next
        current.next = second
        second.next = first

        # Step 2: Move to the next pair
        current = first

    return dummy.next
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

39. Remove Duplicates from Sorted List

Detailed Problem

Given a sorted linked list, delete all duplicates such that each element appears only once.

Input and Output

```
Input: head = [1,1,2]
Output: [1,2]
```

Logic Explanation

We can iterate through the linked list, checking if the current node's value is equal to the next node's value, and adjust the pointers accordingly.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def deleteDuplicates(head: ListNode) -> ListNode:
    current = head

    while current and current.next:
        if current.val == current.next.val:
            current.next = current.next.next  # Skip the duplicate
        else:
            current = current.next  # Move to the next node

    return head
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

40. Remove Duplicates from Sorted List II

Detailed Problem

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers.

Input and Output

```
Input: head = [1,2,3,3,4,4,5]
Output: [1,2,5]
```

Logic Explanation

We can use a dummy node to handle edge cases and use a hashmap to track occurrences of each value. We then skip all nodes with duplicates.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def deleteDuplicates(head: ListNode) -> ListNode:
    dummy = ListNode(0)
    dummy.next = head
    prev = dummy
    current = head

    while current:
        # If it's a duplicate
        if current.next and current.val == current.next.val:
            while current.next and current.val == current.next.val:
                current = current.next
            prev.next = current.next  # Skip duplicates
        else:
            prev = prev.next  # Move to the next distinct node

        current = current.next

    return dummy.next
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

41. Palindrome Linked List

Detailed Problem

Given a singly linked list, determine if it is a palindrome.

Input and Output

```
Input: head = [1,2,2,1]
Output: true
```

Logic Explanation

We can find the middle of the linked list, reverse the second half, and then compare the two halves.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def isPalindrome(head: ListNode) -> bool:
    if not head or not head.next:
        return True

    # Step 1: Find the middle of the linked list
    slow, fast = head, head
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    # Step 2: Reverse the second half of the list
    prev = None
    while slow:
        next_node = slow.next
        slow.next = prev
        prev = slow
        slow = next_node

    # Step 3: Compare the two halves
    left, right = head, prev
    while right:  # Only need to compare to the end of the reversed part
        if left.val != right.val:
            return False
        left = left.next
        right = right.next

    return True
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are using a constant amount of extra space.

---

42. Merge Two Sorted Lists

Detailed Problem

Merge two sorted linked lists and return it as a new sorted list.

Input and Output

```
Input: l1 = [1,2,4], l2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Logic Explanation

We can use a dummy node to simplify the merging process, comparing the heads of both lists and appending the smaller one to the merged list.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode(0)
    current = dummy

    while l1 and l2:
        if l1.val < l2.val:
            current.next = l1
            l1 = l1.next
        else:
            current.next = l2
            l2 = l2.next
        current = current.next

    # Step 3: Append any remaining nodes
    current.next = l1 if l1 else l2

    return dummy.next
```

Time and Space Complexity

- **Time Complexity**: O(n + m), where n and m are the lengths of the two linked lists.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

43. Intersection of Two Linked Lists

Detailed Problem

Given two linked lists, determine if they intersect and return the intersecting node.

Input and Output

```
Input: intersectVal = 8, listA = [4,1,8,4,5], listB = [5,0,1,8,4,5]
Output: 8
```

Logic Explanation

We can calculate the lengths of both lists, align their starting points, and then traverse both lists until we find the intersection.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def getIntersectionNode(headA: ListNode, headB: ListNode) -> ListNode:
    if not headA or not headB:
        return None

    # Step 1: Get lengths
    def getLength(node):
        length = 0
        while node:
            length += 1
            node = node.next
        return length

    lenA, lenB = getLength(headA), getLength(headB)

    # Step 2: Align the start of both lists
    for _ in range(abs(lenA - lenB)):
        if lenA > lenB:
            headA = headA.next
        else:
            headB = headB.next

    # Step 3: Find the intersection
    while headA and headB:
        if headA == headB:
            return headA
        headA = headA.next
        headB = headB.next

    return None
```

Time and Space Complexity

- **Time Complexity**: O(n + m), where n and m are the lengths of the two linked lists.
- **Space Complexity**: O(1), since we are not using any extra data structures.

---

44. Flatten Binary Tree to Linked List

Detailed Problem

Given a binary tree, flatten it to a linked list in place.

Input and Output

```
Input: root = [1,2,5,3,4,null,6]
Output: [1,null,2,null,3,null,4,null,5,null,6]
```

Logic Explanation

We can perform a pre-order traversal of the binary tree, modifying the nodes as we go.

Code

```python
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def flatten(root: TreeNode) -> None:
    if not root:
        return

    # Step 1: Flatten the left and right subtrees
    flatten(root.left)
    flatten(root.right)

    # Step 2: Reassign the pointers
    temp_right = root.right
    root.right = root.left
    root.left = None

    # Step 3: Find the rightmost node of the new right subtree
    while root.right:
        root = root.right

    root.right = temp_right  # Connect the right subtree
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the binary tree.
- **Space Complexity**: O(h), where h is the height of the tree due to the recursion stack.

---

45. Convert Sorted List to Binary Search Tree

Detailed Problem

Given a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Input and Output

```
Input: head = [-10,-3,0,5,9]
Output: [0,-10,5,null,-3,null,9]
```

Logic Explanation

We can use a recursive approach to divide the linked list into sublists, constructing nodes for the BST as we go.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def sortedListToBST(head: ListNode) -> TreeNode:
    if not head:
        return None

    # Step 1: Find the middle node
    def find_middle(left, right):
        slow = left
        fast = left
        while fast != right and fast.next != right:
            slow = slow.next
            fast = fast.next.next
        return slow

    # Step 2: Recursively build the tree
    def convert(left, right):
        if left == right:
            return None
        mid = find_middle(left, right)
        node = TreeNode(mid.val)
        node.left = convert(left, mid)
        node.right = convert(mid.next, right)
        return node

    return convert(head, None)
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(log n), due to the recursion stack in a balanced tree.

---

46. Find the Duplicate Number

Detailed Problem

Given an array of integers containing `n + 1` integers where each integer is between `1` and `n`, there is only one duplicate number. Find the duplicate.

Input and Output

```
Input: nums = [1,3,4,2,2]
Output: 2
```

Logic Explanation

We can use the Floyd's Tortoise and Hare cycle detection algorithm to find the duplicate number.

Code

```python
def findDuplicate(nums):
    # Step 1: Use two pointers to find the intersection point
    slow = nums[0]
    fast = nums[0]

    while True:
        slow = nums[slow]
        fast = nums[nums[fast]]
        if slow == fast:
            break

    # Step 2: Find the entrance to the cycle
    slow = nums[0]
    while slow != fast:
        slow = nums[slow]
        fast = nums[fast]

    return slow
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of elements in the array.
- **Space Complexity**: O(1), since we are using a constant amount of space.

---

47. Reverse Nodes in k-Group

Detailed Problem

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

Input and Output

```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
```

Logic Explanation

We can reverse nodes in groups of k, using a dummy node to keep track of the head of the reversed list.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def reverseKGroup(head: ListNode, k: int) -> ListNode:
    dummy = ListNode(0)
    dummy.next = head
    prev_group_end = dummy

    while True:
        kth_node = prev_group_end
        for _ in range(k):
            kth_node = kth_node.next
            if not kth_node:
                return dummy.next

        group_start = prev_group_end.next
        group_end = kth_node.next
        kth_node.next = None

        # Reverse the group
        prev, curr = None, group_start
        while curr:
            next_node = curr.next
            curr.next = prev
            prev = curr
            curr = next_node

        prev_group_end.next = prev
        group_start.next = group_end
        prev_group_end = group_start
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.

---

48. Copy List with Random Pointer

Detailed Problem

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null. You need to return a deep copy of the list.

Input and Output

```
Input: head = [[7,null],[13,0],[11,4],[10,2],[1,0]]
Output: [[7,null],[13,0],[11,4],[10,2],[1,0]]
```

Logic Explanation

We can create a mapping of original nodes to their copies and then set the random pointers accordingly.

Code

```python
class RandomListNode:
    def __init__(self, label, next=None, random=None):
        self.label = label
        self.next = next
        self.random = random

def copyRandomList(head: RandomListNode) -> RandomListNode:
    if not head:
        return None

    # Step 1: Create a copy of each node and link them
    current = head
    while current:
        copy = RandomListNode(current.label)
        copy.next = current.next
        current.next = copy
        current = copy.next

    # Step 2: Set the random pointers
    current = head
    while current:
        if current.random:
            current.next.random = current.random.next
        current = current.next.next

    # Step 3: Separate the original list from the copied list
    current = head
    copy_head = head.next
    while current:
        copy = current.next
        current.next = copy.next
        current = current.next
        if copy:
            copy.next = current.next if current else None

    return copy_head
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(n), for the copies of the nodes.

---

49. Palindrome Linked List II

Detailed Problem

Determine if the linked list has a palindrome sublist.

Input and Output

```
Input: head = [1,2,3,4,3,2,1]
Output: true
```

Logic Explanation

We can check all possible sublists using a two-pointer technique to find potential palindromes.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def isPalindromeSublist(head: ListNode) -> bool:
    def isPalindrome(start: ListNode, end: ListNode) -> bool:
        while start and end:
            if start.val != end.val:
                return False
            start = start.next
            end = end.next
        return True

    current = head
    while current:
        # Check for every sublist starting from current
        temp = current
        while temp:
            if isPalindrome(current, temp):
                return True
            temp = temp.next
        current = current.next

    return False
```

Time and Space Complexity

- **Time Complexity**: O(n^2), where n is the number of nodes in the linked list.

- **Space Complexity**: O(1), since we are not using any additional data structures.

---

50. Odd Even Linked List

Detailed Problem

Given a linked list, group all odd nodes together followed by the even nodes. The odd and even nodes should maintain their relative order.

Input and Output

```
Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]
```

Logic Explanation

We can maintain two pointers for odd and even nodes and then link them at the end.

Code

```python
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def oddEvenList(head: ListNode) -> ListNode:
    if not head or not head.next:
        return head

    odd = head
    even = head.next
    even_head = even

    while even and even.next:
        odd.next = odd.next.next  # Link odd nodes
        even.next = even.next.next  # Link even nodes
        odd = odd.next
        even = even.next

    odd.next = even_head  # Connect odd list with even list

    return head
```

Time and Space Complexity

- **Time Complexity**: O(n), where n is the number of nodes in the linked list.
- **Space Complexity**: O(1), since we are modifying the list in place.
