Linked list

1. Merge Two Sorted Lists (Easy)

Problem Statement

You are given the heads of two sorted linked lists `list1` and `list2`. Merge the two lists into one sorted linked list. Return the head of the merged linked list.

Sample Input

```
list1 = [1, 2, 4]
list2 = [1, 3, 4]
```

Sample Output

```
Output: [1, 1, 2, 3, 4, 4]
```

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeTwoLists(self, list1, list2):
        """
        Merges two sorted singly linked lists into a single sorted
        linked list.

        Args:
          list1: The first sorted linked list.
```

```
          list2: The second sorted linked list.

        Returns:
          A new sorted linked list containing all elements from
          list1 and list2.
        """

        # Create a dummy node to simplify handling the head of
        # the merged list
        dummy = ListNode(-1)
        current = dummy

        # Iterate while both lists have elements remaining
        while list1 and list2:
            # Choose the smaller element and add it to the
            # merged list
            if list1.val < list2.val:
                current.next = list1
                list1 = list1.next
            else:
                current.next = list2
                list2 = list2.next
            # Move the current pointer to the newly added node
            current = current.next

        # Append the remaining elements from either list (if any)
        current.next = list1 or list2

        # Return the actual start of the merged list
        # (excluding the dummy node)
        return dummy.next

# Test case
list1 = ListNode(1, ListNode(2, ListNode(4)))
list2 = ListNode(1, ListNode(3, ListNode(4)))

merged_list = Solution().mergeTwoLists(list1, list2)

# Print the elements of the merged list (optional)
while merged_list:
    print(merged_list.val, end=" -> ")
    merged_list = merged_list.next
```

```
print("None")

list3 = None
list4 = ListNode(0)
merged_list2 = Solution().mergeTwoLists(list3, list4)

while merged_list2:
    print(merged_list2.val, end=" -> ")
    merged_list2 = merged_list2.next
print("None")
```

```
1 -> 1 -> 2 -> 3 -> 4 -> 4 -> None
0 -> None
```

Understanding the Problem

The goal is to merge two sorted singly linked lists (`list1` and `list2`) into a single sorted linked list.

Step-by-Step Explanation of the Code

Class Definitions

1. **`ListNode` Class**:
   - Represents a node in a singly linked list.
   - Each node contains a value (`val`) and a reference to the next node (`next`).
2. **`Solution` Class**:
   - Contains the method `mergeTwoLists`, which merges two sorted linked lists.

`mergeTwoLists` Method

1. **Create a Dummy Node**:

```
dummy = ListNode(-1)
current = dummy
```

- A dummy node simplifies handling the head of the merged list.
- `current` is used to traverse and build the merged list.

2. **Merge Two Lists**:

```
while list1 and list2:
```

- Continue iterating while both lists have remaining elements.
- At each step, compare the current nodes' values (`list1.val` and `list2.val`).
- **Add the Smaller Node**:

```
if list1.val < list2.val:
    current.next = list1
    list1 = list1.next
else:
    current.next = list2
    list2 = list2.next
current = current.next
```

  - Add the smaller node to the merged list.
  - Move the pointer (`list1` or `list2`) forward.
  - Advance the `current` pointer to continue building the list.

3. **Attach Remaining Nodes**:

```
current.next = list1 or list2
```

- After the loop, one list may still have remaining nodes.
- Append these nodes directly to the merged list.

4. **Return the Merged List**:

```
return dummy.next
```

- Exclude the dummy node and return the actual start of the merged list.

Example Walkthrough

Example Input:

```
list1 = 1 -> 2 -> 4
list2 = 1 -> 3 -> 4
```

Execution:

1. Initialize:
   - `dummy = -1`
   - `current = dummy`
2. **Step 1**:
   - Compare `list1.val (1)` and `list2.val (1)`.
   - Append `list1` or `list2` (both have the same value) to the merged list.
   - `current = 1`.

   Result: `dummy -> 1`.
3. **Step 2**:
   - Compare `list1.val (2)` and `list2.val (3)`.
   - Append `list1` (smaller value) to the merged list.
   - `current = 2`.

   Result: `dummy -> 1 -> 2`.
4. **Step 3**:
   - Compare `list1.val (4)` and `list2.val (3)`.
   - Append `list2` (smaller value) to the merged list.
   - `current = 3`.

   Result: `dummy -> 1 -> 2 -> 3`.
5. **Step 4**:
   - Compare `list1.val (4)` and `list2.val (4)`.
   - Append `list1` or `list2` (both have the same value) to the merged list.
   - `current = 4`.

Result: `dummy -> 1 -> 2 -> 3 -> 4`.

6. **Attach Remaining Nodes**:
   - `list1` still has one node (`4`). Append it.
   - `current = 4`.

   Final Result: `dummy -> 1 -> 1 -> 2 -> 3 -> 4 -> 4`.

Output:

`1 -> 1 -> 2 -> 3 -> 4 -> 4`.

Complexity Analysis

1. **Time Complexity**:
   - Each element is visited once.
   - Total time: (O(m + n)), where (m) and (n) are the lengths of `list1` and `list2`.
2. **Space Complexity**:
   - No additional space is used except for the dummy node.
   - Space: (O(1)).

---

2. Middle of the Linked List (Easy)

Problem Statement

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

Example 1:

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

Output:
`[3, 4, 5]`

Explanation:
The middle node is `3`.

Example 2:

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

Output:
`[4, 5, 6]`

Explanation:
Since the list has two middle nodes (`3` and `4`), the second one is considered as the middle node.

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

def middleNode(head):
    # Initialize two pointers: slow and fast
    slow = head  # Moves one step at a time
    fast = head  # Moves two steps at a time

    # Traverse the list
    while fast is not None and fast.next is not None:
        slow = slow.next          # Move slow pointer one step
        fast = fast.next.next     # Move fast pointer two steps

    # When the loop ends, slow is at the middle of the list
    return slow

# Helper function to create a linked list from a list
def create_linked_list(values):
    head = ListNode(values[0])  # Create the head node
    current = head
```

```
    for value in values[1:]:
        current.next = ListNode(value)  # Create the next node
        current = current.next
    return head

# Helper function to print a linked list
def print_linked_list(head):
    result = []
    while head:
        result.append(head.val)
        head = head.next
    return result

# Sample Test Case
values = [1, 2, 3, 4, 5]  # Input list
head = create_linked_list(values)  # Create linked list from input
print("Original Linked List:", print_linked_list(head))

# Find and print the middle node
middle = middleNode(head)
print("Middle Node Value:", middle.val)
```

```
Original Linked List: [1, 2, 3, 4, 5]
Middle Node Value: 3
```

Time Complexity: O(n) - The fast pointer traverses the list at twice the speed of the slow pointer, resulting in a single pass through the list.
Space Complexity: O(1) - Only a constant amount of extra space is used for the two pointers.

3. Linked List Cycle (Easy)

Problem Statement

Given the head of a linked list, determine if the linked list has a cycle in it. A linked list has a cycle if there is some node in the list that can be reached again by continuously following the `next` pointer.

Sample Input

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

Sample Output

```
Output: true
```

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

def hasCycle(head):
    # Initialize two pointers: slow moves one step, fast moves two steps
    slow, fast = head, head

    while fast and fast.next:
        slow = slow.next         # Move slow pointer one step
        fast = fast.next.next    # Move fast pointer two steps

        if slow == fast:         # Cycle detected
            return True

    return False  # If we exit the loop, no cycle exists

# Sample Test Case 1
# Input: 3 -> 2 -> 0 -> -4 -> (points back to node 2)
# Output: True
node1 = ListNode(3)
node2 = ListNode(2)
node3 = ListNode(0)
node4 = ListNode(-4)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node2  # Creates a cycle

print(hasCycle(node1))  # Output: True

# Sample Test Case 2
```

```
# Input: 1 -> 2 -> None
# Output: False
node1 = ListNode(1)
node2 = ListNode(2)
node1.next = node2

print(hasCycle(node1))  # Output: False
```

```
True
False
```

- **Time Complexity**: O(n), where n is the number of nodes in the linked list, as each node is visited at most once.
- **Space Complexity**: O(1), as only two pointers (slow and fast) are used, requiring constant space.

4. Reverse Linked List (Easy)

Problem Statement

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

Sample Input

```
head = [1, 2, 3, 4, 5]
```

Sample Output

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

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

def reverseList(head):
    # Initialize two pointers: `prev` as None and
    # `current` as the head of the list.
    prev = None
```

```
    current = head

    while current:
        # Store the next node before reversing the link
        next_node = current.next

        # Reverse the link
        current.next = prev

        # Move `prev` and `current` one step forward
        prev = current
        current = next_node

    # `prev` will be the new head of the reversed list
    return prev

# Sample Test Case
# Input: 1 -> 2 -> 3 -> 4 -> 5 -> None
# Output: 5 -> 4 -> 3 -> 2 -> 1 -> None
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
new_head = reverseList(head)

# Print reversed list
curr = new_head
while curr:
    print(curr.val, end=" -> ")
    curr = curr.next
# Output: 5 -> 4 -> 3 -> 2 -> 1 -> None
```

```
5 -> 4 -> 3 -> 2 -> 1 ->
```

Time Complexity: O(n), where n is the number of nodes in the linked list, as we traverse the list once.
Space Complexity: O(1), as we reverse the list in place without using additional memory.

5. Palindrome Linked List (Easy)

Problem Statement

Given the head of a singly linked list, return `true` if it is a palindrome.

Sample Input

```
head = [1, 2, 2, 1]
```

Sample Output

```
Output: true
```

```
def isPalindrome(head):
    # Step 1: Find the middle of the linked list using the
    # slow-fast pointer approach
    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:  # Right half is shorter or equal to the left
        if left.val != right.val:
            return False
        left = left.next
        right = right.next

    return True

# Sample Test Case
# Input: 1 -> 2 -> 2 -> 1 -> None
# Output: True
head = ListNode(1, ListNode(2, ListNode(2, ListNode(1))))
print(isPalindrome(head))  # Output: True
```

```
True
```

Time Complexity: O(n), where n is the number of nodes in the linked list (traversed twice: once to find the middle and once to compare halves).
Space Complexity: O(1), as the reversal of the list is done in place without using extra space.

6. Reverse Nodes in k-Group (Hard)

Problem Statement

Given the head of a linked list, reverse the nodes of the list `k` at a time, and return the modified list. `k` is a positive integer, and it is guaranteed that the number of nodes in the list is a multiple of `k`.

Sample Input

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

Sample Output

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

```python
def reverseKGroup(head, k):
    # Helper function to reverse a sublist
    def reverse_sublist(start, end):
        prev = None
        current = start
        while current != end:
            next_node = current.next
            current.next = prev
            prev = current
            current = next_node
        return prev

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

    while True:
        # Find the kth node from `group_prev`
        kth = group_prev
```

```
        for _ in range(k):
            kth = kth.next
            if not kth:  # If fewer than k nodes remain
                return dummy.next

        group_next = kth.next  # Store the node after the kth node
        # Reverse the group of k nodes
        prev = reverse_sublist(group_prev.next, group_next)

        # Adjust the connections
        start = group_prev.next  # The start becomes the end after reversal
        group_prev.next = prev
        start.next = group_next

        # Move `group_prev` to the end of the reversed group
        group_prev = start

# Sample Test Case
# Input: 1 -> 2 -> 3 -> 4 -> 5 -> None, k = 2
# Output: 2 -> 1 -> 4 -> 3 -> 5 -> None
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
k = 2
new_head = reverseKGroup(head, k)

# Print reversed in k-groups
curr = new_head
while curr:
    print(curr.val, end=" -> ")
    curr = curr.next
# Output: 2 -> 1 -> 4 -> 3 -> 5 -> None
```

```
2 -> 1 -> 4 -> 3 -> 5 ->
```

Time Complexity: O(n), where n is the number of nodes in the linked list, as each node is processed at most twice (once during traversal and once during reversal).
Space Complexity: O(1), as the reversal is done in-place without using extra space except for a few pointers.

7. Remove Nth Node From End of List (Medium)

Problem Statement

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

Sample Input

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

Sample Output

```
Output: [1, 2, 3, 5]
```

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

def removeNthFromEnd(head, n):
    # Create a dummy node to handle edge cases like removing the first node.
    dummy = ListNode(0)
    dummy.next = head

    # Initialize two pointers: first and second.
    first = dummy
    second = dummy

    # Move the first pointer `n+1` steps forward so there's a gap of `n`
    # nodes between first and second.
    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

    # Now, the second pointer is just before the node to be removed.
    second.next = second.next.next  # Remove the nth node.

    return dummy.next  # Return the updated list, skipping the dummy node.

# Sample Test Case
```

```
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
n = 2
result = removeNthFromEnd(head, n)

# Print the result list
while result:
    print(result.val, end=" -> ")
    result = result.next
# Output: 1 -> 2 -> 3 -> 5 ->
```

```
1 -> 2 -> 3 -> 5 ->
```

Time Complexity: O(L), where L is the length of the linked list, as we traverse the list twice (once to move the first pointer and once to adjust the second pointer).
Space Complexity: O(1), as no additional space is used apart from a few pointers.

8. Odd Even Linked List (Medium)

Problem Statement

Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

Sample Input

```
head = [1, 2, 3, 4, 5]
```

Sample Output

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

```
def oddEvenList(head):
    if not head:
        return None  # If the list is empty, return None.

    # Initialize two pointers for odd and even nodes.
    odd = head
    even = head.next
    even_head = even  # Keep the head of the even list to connect later.
```

```
    # Traverse the list, separating odd and even nodes.
    while even and even.next:
        odd.next = even.next  # Link current odd node to the next odd node.
        odd = odd.next  # Move the odd pointer forward.
        even.next = odd.next  # Link current even node to the next even node.
        even = even.next  # Move the even pointer forward.

    # After traversal, connect the last odd node to the head of the even list.
    odd.next = even_head

    return head  # Return the updated list.

# Sample Test Case
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
result = oddEvenList(head)

# Print the result list
while result:
    print(result.val, end=" -> ")
    result = result.next
# Output: 1 -> 3 -> 5 -> 2 -> 4 ->
```

```
1 -> 3 -> 5 -> 2 -> 4 ->
```

Time Complexity: O(n) - Each node is visited once.
Space Complexity: O(1) - The rearrangement is done in place without using additional space.

9. Swap Nodes in Pairs (Medium)

Problem Statement

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed).

Sample Input

```
head = [1, 2, 3, 4]
```

Sample Output

```
Output: [2, 1, 4, 3]
```

```
def swapPairs(head):
    # Create a dummy node to simplify edge case handling.
    dummy = ListNode(0)
    dummy.next = head
    current = dummy

    # Traverse the list and swap pairs.
    while current.next and current.next.next:
        first = current.next  # First node of the pair.
        second = current.next.next  # Second node of the pair.

        # Perform the swap.
        first.next = second.next
        second.next = first
        current.next = second

        # Move to the next pair.
        current = first

    return dummy.next  # Return the updated list.

# Sample Test Case
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))
result = swapPairs(head)

# Print the result list
while result:
    print(result.val, end=" -> ")
    result = result.next
# Output: 2 -> 1 -> 4 -> 3 ->
```

```
2 -> 1 -> 4 -> 3 ->
```

Time Complexity: O(n), where n is the number of nodes in the linked list, as we traverse each node once.
Space Complexity: O(1), as we perform the swaps in place without using additional data structures.

10. Reorder List (Medium)

Problem Statement

You are given the head of a singly linked list. Reorder the list to follow the pattern: `L0 -> Ln -> L1 -> Ln-1 -> L2 -> Ln-2 -> ...`

Sample Input

```
head = [1, 2, 3, 4, 5]
```

Sample Output

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

```python
def reorderList(head):
    if not head or not head.next or not head.next.next:
        return  # If the list is too short, no reordering needed.

    # Step 1: Find the middle of the list using the slow and
    # fast pointer approach.
    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.next
    slow.next = None  # Split the list into two halves.
    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:
        tmp1, tmp2 = first.next, second.next
        first.next = second
        second.next = tmp1
```

```
        first, second = tmp1, tmp2

# Sample Test Case
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
reorderList(head)

# Print the result list
result = head
while result:
    print(result.val, end=" -> ")
    result = result.next
# Output: 1 -> 5 -> 2 -> 4 -> 3 ->
```

```
1 -> 5 -> 2 -> 4 -> 3 ->
```

Time Complexity: O(N), where N is the number of nodes in the linked list (finding the middle, reversing, and merging each take O(N)).
Space Complexity: O(1), as the reordering is done in-place without using additional data structures.

11. Add Two Numbers (Medium)

Problem Statement

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 the sum as a linked list.

Sample Input

```
l1 = [2, 4, 3]
l2 = [5, 6, 4]
```

Sample Output

```
Output: [7, 0, 8]
```

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

def addTwoNumbers(l1: ListNode, l2: ListNode) -> ListNode:
    # Dummy node to simplify edge cases
    dummy = ListNode()
    current = dummy
    carry = 0

    # Iterate through both linked lists
    while l1 or l2 or carry:
        # Extract values, defaulting to 0 if the node is None
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        # Calculate sum and carry
        total = val1 + val2 + carry
        carry = total // 10  # Update carry for the next digit
        # Create a new node with the single-digit sum
        current.next = ListNode(total % 10)

        # Move to the next nodes
        current = current.next
        if l1: l1 = l1.next
        if l2: l2 = l2.next

    return dummy.next

# Sample Test Case
# l1 = [2 -> 4 -> 3], l2 = [5 -> 6 -> 4]
l1 = ListNode(2, ListNode(4, ListNode(3)))
l2 = ListNode(5, ListNode(6, ListNode(4)))
result = addTwoNumbers(l1, l2)
# Expected output: [7 -> 0 -> 8]
while result:
    print(result.val, end=" -> " if result.next else "\n")
    result = result.next
```

```
7 -> 0 -> 8
```

Time Complexity: O(max(m, n)), where m and n are the lengths of the two linked lists.
Space Complexity: O(max(m, n)), for storing the result linked list.

12. Rotate List (Medium)

Problem Statement

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

Sample Input

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

Sample Output

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

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

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

    # Step 2: Find the new head position
    k = k % length
    if k == 0:  # No rotation needed
        return head

    # Step 3: Make the list circular
    tail.next = head

    # Step 4: Find the new tail (length - k - 1 moves from head)
    new_tail = head
    for _ in range(length - k - 1):
        new_tail = new_tail.next
```

```
    # Step 5: Break the circular list to form the new list
    new_head = new_tail.next
    new_tail.next = None

    return new_head

# Sample Test Case
# head = [1 -> 2 -> 3 -> 4 -> 5], k = 2
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
result = rotateRight(head, 2)
# Expected output: [4 -> 5 -> 1 -> 2 -> 3]
while result:
    print(result.val, end=" -> " if result.next else "\n")
    result = result.next
```

```
4 -> 5 -> 1 -> 2 -> 3
```

Time Complexity: O(n), where n is the number of nodes in the linked list, as we traverse the list to calculate its length and perform rotations.

Space Complexity: O(1), since we are using a constant amount of extra space, aside from the input list.

13. Sort List (Medium)

Problem Statement

Given the head of a linked list, return the list after sorting it in ascending order.

Sample Input

```
head = [4, 2, 1, 3]
```

Sample Output

```
Output: [1, 2, 3, 4]
```

```python
def sortList(head: ListNode) -> ListNode:
    # Base case for recursion
    if not head or not head.next:
        return head

    # Step 1: Split the list into two halves
    slow, fast = head, head.next
    while fast and fast.next:
        slow = slow.next
        fast = fast.next.next

    mid = slow.next
    slow.next = None

    # Step 2: Recursively sort each half
    left = sortList(head)
    right = sortList(mid)

    # Step 3: Merge the sorted halves
    return merge(left, right)

def merge(l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode()
    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

    # Append remaining nodes
    current.next = l1 if l1 else l2
    return dummy.next

# Sample Test Case
# head = [4 -> 2 -> 1 -> 3]
head = ListNode(4, ListNode(2, ListNode(1, ListNode(3))))
result = sortList(head)
```

```
# Expected output: [1 -> 2 -> 3 -> 4]
while result:
    print(result.val, end=" -> " if result.next else "\n")
    result = result.next
```

```
1 -> 2 -> 3 -> 4
```

Time Complexity: O(n log n), where n is the number of nodes in the list, due to the divide and conquer approach (merge sort).

Space Complexity: O(log n), due to the recursive call stack used during the merge sort process.

14. LRU Cache (Medium)

Problem Statement

Design a data structure that follows the constraints of a **Least Recently Used (LRU) Cache**. Implement the `LRUCache` class with the following methods: - `get(key)`: Return the value of the key if the key exists, otherwise return -1. - `put(key, value)`: Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache.

Sample Input

```
LRUCache cache = new LRUCache(2);
cache.put(1, 1);
cache.put(2, 2);
cache.get(1);
cache.put(3, 3);
cache.get(2);
cache.put(4, 4);
cache.get(1);
cache.get(3);
cache.get(4);
```

Sample Output

```
Output: [1, -1, -1, 3, 4]
```

```python
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity: int):
        self.cache = OrderedDict()
        self.capacity = capacity

    def get(self, key: int) -> int:
        if key in self.cache:
            # Move the accessed key to the end (most recently used)
            self.cache.move_to_end(key)
            return self.cache[key]
        return -1  # Key not found

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            # Update the value and move to the end
            self.cache.move_to_end(key)
        elif len(self.cache) >= self.capacity:
            # Remove the least recently used item
            self.cache.popitem(last=False)
        self.cache[key] = value  # Insert the key-value pair

# Sample Test Case
cache = LRUCache(2)  # Capacity = 2
cache.put(1, 1)  # Cache: {1: 1}
cache.put(2, 2)  # Cache: {1: 1, 2: 2}
print(cache.get(1))  # Expected output: 1, Cache: {2: 2, 1: 1}
cache.put(3, 3)  # Cache: {1: 1, 3: 3} (2 is evicted)
print(cache.get(2))  # Expected output: -1 (not found)
cache.put(4, 4)  # Cache: {3: 3, 4: 4} (1 is evicted)
print(cache.get(1))  # Expected output: -1 (not found)
print(cache.get(3))  # Expected output: 3
print(cache.get(4))  # Expected output: 4
```

```
1
-1
-1
3
4
```

Time Complexity:

- `get(key)` and `put(key, value)` both have an average time complexity of O(1) due to the use of `OrderedDict`.

Space Complexity:
- O(capacity), as the space is used to store up to `capacity` key-value pairs in the cache.