24. Swap Nodes in Pairs

**Linked List** 

employ two main strategies: recursion or iteration.

## **Problem Description**

Recursion

Medium

Given a singly linked list, the objective is to swap every two adjacent nodes and return the modified linked list. Unlike simple value swapping, this task requires changing the actual node connections. Importantly, the modification must be carried out without altering the values within the nodes -- in essence, only the node linkages can be reshuffled. The problem also asks for the function to accommodate linked lists with an odd number of nodes, in which case the final node remains in its original position since there's no pair to swap with.

Intuition

Recursive Approach: We perform a depth-first traversal, recursively swapping pairs of nodes. In each recursive call, we handle a pair and then delve deeper, assuming that the rest of the list beyond this pair will be solved in the same recursive manner. When the <u>recursion</u> unwinds, the entire list is thus reordered in pairs.

Transforming the structure of a <u>linked list</u> often points towards manipulating node pointers. To swap nodes in pairs, we can

- Iterative Approach: Iteration is the alternative strategy that seems natural for this problem, as linked list manipulation often involves loop constructs. The provided code snippet uses an iterative approach with two pointers, which sidesteps the
- complexities that can come with <u>recursion</u>, such as stack space concerns. We introduce a dummy node that precedes the head of the list for simplicity, allowing us to standardize the swapping process without special-casing the head of the list. Two pointers pre and cur are established to maintain references to the nodes being operated on. cur points at the current node under consideration and pre trails behind, enabling us to properly link the swapped pairs to the rest of the list. By looping through the list and updating these pointers step-by-step, we systematically exchange the adjacent nodes, while preserving the original node values and ensuring all connections are accurately reestablished post-swap.
- **Solution Approach** This problem can be resolved by two approaches, <u>recursion</u> or iteration, using the fundamental concepts of <u>linked list</u> traversal and pointer manipulation.

# recursively call the function on the sublist starting with the third node. Here's how the process works:

**Solution 1: Recursion** 

1. The base condition checks if the current node (head) is null or if it's the last node with no pair to swap (head next is null). In either case, the head is returned as-is, since no more swapping is possible.

2. For any node with at least one adjacent node to swap, we store the next node in t and perform the swap by setting head next to the recursive

The recursive approach to the problem often provides a clean and intuitive solution. Here, we swap each pair of nodes and

#### call on head.next.next. This effectively links the current head to the result of swapping the rest of the list. 3. The second node of the pair (t) now needs to point to the first (head), forming the swapped pair, and t is returned as the new head of this

swapped section. Each step handles two nodes and links the swapped pair to the result of the rest of the list, which is computed recursively.

- The time complexity of this approach is 0(n), where n is the number of nodes in the linked list, as each node pair is visited once. The space complexity is also "O(n)" on account of the recursive call stack.
- **Solution 2: Iteration** The iterative solution approach is generally more space-efficient for linked lists as it avoids the overhead of the recursive call stack.

as any other node in the swapping process, which simplifies the code. 2. We then set two pointers, pre starting at the dummy node, and cur starting at the list's actual head.

1. We start by creating a dummy node that acts as a placeholder prior to the head of the list. This allows us to handle the head of the list the same

3. Within a loop, as long as cur and its next node cur.next are not null, we perform the swap. We first store the next node in t, then link cur.next

### 5. We now reset the pointer pre.next to t to connect the previous part of the list to our newly swapped pair. 6. Finally, we update pre to cur (the first node of the swapped pair) and move cur to cur. next to process the next pair.

end of the list.

solving the problem at hand.

[dummy  $\rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5$ ]

the real list (node with value 1).

**Example Walkthrough** 

 $[1 \rightarrow 2 \rightarrow 3 \rightarrow 4 \rightarrow 5]$ 

provided.

4. After that, t.next is updated to point to cur, effectively placing t before cur.

Let's assume we have a linked list with the following values:

First, we make cur.next point to t.next (which is node 3):

Then we update t.next to point to cur, placing t before cur:

to t.next.

is improved to "0(1)" because we aren't making any recursive calls, just reassigning pointers. Both approaches effectively change node pointers to swap adjacent nodes in pairs without altering the node values, successfully

The time complexity for this iterative approach is also "O(n)" since each node is visited exactly once, while the space complexity

With this pattern, we successively swap adjacent nodes and move forward in the list until all pairs are swapped or we reach the

Our goal is to swap every pair of adjacent nodes. We'll walk through the iterative approach since it's mentioned in the content

We introduce a dummy node to simplify our process. The list now starts with a dummy node followed by the original nodes:

Here, dummy is a standalone node that we use as a starting point, pre starts as the dummy node, and cur starts at the head of

We check if cur and cur.next (nodes 1 and 2) are not null, to proceed with the swap. Since they are both not null, we move onto swapping the nodes. Let's denote t as the node after cur, which is node 2. We swap the nodes by reassigning pointers:

# [dummy → 1 ---

2 → 1 ----

 $3 \rightarrow 4 \rightarrow 5$ 

[dummy  $\rightarrow$  2  $\rightarrow$  1 ---

After swapping, we reconnect the list:

We update pre to cur (node 3) and cur to cur.next (node 5).

[dummy  $\rightarrow$  2  $\rightarrow$  1  $\rightarrow$  4  $\rightarrow$  3  $\rightarrow$  5]

4 → 3 ---

5] [3 ---/

 $[2 \rightarrow 1 \rightarrow 4 \rightarrow 3 \rightarrow 5]$ 

**Python** 

class ListNode:

class Solution:

Java

C++

/\*\*

Solution Implementation

# Definition for singly-linked list.

self.next = next\_node

self.val = val

def \_\_init\_\_(self, val=0, next\_node=None):

dummy = ListNode(next\_node=head)

temp.next = current\_node

ListNode previousNode = dummyNode;

ListNode currentNode = head;

prev\_node.next = temp

return dummy.next

prev\_node, current\_node = dummy, head

 $3 \to 4 \to 5$ ] [2 ---/

[dummy ---

Now we link the previous part of the list (the dummy node) to our swapped pair by setting pre.next to t:

[dummy  $\rightarrow$  2  $\rightarrow$  1  $\rightarrow$  3  $\rightarrow$  4  $\rightarrow$  5] We then advance our pointers: pre moves to cur (node 1), and cur moves to cur.next (node 3).

We continue with the loop, checking if there are more pairs to swap. Again, cur and cur. next are not null, so we store the

Lastly, we find that cur. next is null because node 5 has no pair to swap with. Therefore, we stop the loop and return the head

By following these steps iteratively, we've managed to swap the adjacent nodes in pairs throughout the list while maintaining the

original values, just as required by the problem statement. The time complexity of this operation is O(n) since we visited each

of the modified list, which is the node immediately following our dummy node. The final swapped list is:

node after cur (t, which is node 4) and perform a similar set of pointer reassignments:

node once, and the space complexity is 0(1) due to the constant number of pointers used.

# 'temp' is the node that will be swapped with the current node temp = current\_node.next # Adjust 'current\_node.next' to the node after 'temp' current\_node.next = temp.next

while current\_node and current\_node.next:

# Return the new head of the swapped list

def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:

# The 'temp' node now should point to 'current\_node' after swapping

// 'previousNode' always points to the node before the pair that needs to be swapped.

// Move 'previousNode' pointer two nodes ahead to the last node of the swapped pair.

// 'currentNode' is the first node in the pair that needs to be swapped.

// Advance 'currentNode' to the next pair of nodes to swap.

// The 'next' of dummy node points to the new head after swapping pairs.

// 'nextNode' is the second node in the pair that needs to be swapped.

// Iterate over the list in steps of two nodes at a time.

while (currentNode != null && currentNode.next != null) {

ListNode nextNode = currentNode.next;

currentNode.next = nextNode.next;

nextNode.next = currentNode;

previousNode = currentNode;

previousNode = currentNode;

currentNode = currentNode->next;

ListNode\* swappedHead = dummyNode->next;

// Type definition for a singly-linked list node.

// Helper function to create a new ListNode.

return { val: val, next: next };

// Traverse the list in pairs.

while (current && current.next) {

let temp = current.next;

current.next = temp.next;

temp.next = current;

previous.next = temp;

previous = current;

current = current.next;

prev\_node.next = temp

return dummy.next

Time and Space Complexity

# Return the new head of the swapped list

// Store the node following the current node.

// Skip the next node to point to the one after.

// Move the pointers forward to the next pair.

// Return the new head, which is the next of the dummy node.

delete dummyNode;

return swappedHead;

**}**;

**}**;

**TypeScript** 

type ListNode = {

val: number;

next: ListNode | null;

// Clean up memory by deleting the dummy node

return dummyNode.next;

\* Definition for singly-linked list.

\* struct ListNode {

currentNode = currentNode.next;

previousNode.next = nextNode;

// Swap the pair by adjusting the pointers.

# 'prev\_node.next' is adjusted to point to 'temp' after swapping

# Move 'prev\_node' and 'current\_node' forward in the list

prev\_node, current\_node = current\_node, current\_node.next

# Create a dummy node that points to the head of the list

# Iterate through the list while there are pairs to swap

```
/**
* Definition for singly-linked list.
* public class ListNode {
      int val;
      ListNode next;
      ListNode() {}
      ListNode(int val) { this.val = val; }
*
      ListNode(int val, ListNode next) { this.val = val; this.next = next; }
*
* }
*/
class Solution {
    public ListNode swapPairs(ListNode head) {
       // The dummy node is used to simplify the edge case where the list might contain only one node.
       ListNode dummyNode = new ListNode(0);
       dummyNode.next = head;
```

```
int val;
      ListNode *next;
      ListNode(int x) : val(x), next(nullptr) {}
      ListNode(int x, ListNode *next) : val(x), next(next) {}
      ListNode(): val(0), next(nullptr) {}
* };
*/
class Solution {
public:
   ListNode* swapPairs(ListNode* head) {
       // Create a dummy node to anchor the modified list and simplify edge cases
       ListNode* dummyNode = new ListNode(0);
       dummyNode->next = head;
       // Use 'previousNode' to keep track of the last node of the previous pair
       ListNode* previousNode = dummyNode;
       // 'currentNode' will be used to iterate through the original list
       ListNode* currentNode = head;
       // Proceed with swapping pairs while there are at least two nodes left to process
       while (currentNode && currentNode->next) {
            // Identify the node to be swapped with the current node
            ListNode* nextNode = currentNode->next;
           // Swap the pair by reassigning pointers
            currentNode->next = nextNode->next;
            nextNode->next = currentNode;
            previousNode->next = nextNode;
```

// Move 'previousNode' pointer forward to the current node after swap

// Move 'currentNode' pointer forward to the next pair

// Return the head of the modified list with pairs swapped

// The 'dummyNode.next' now points to the head of the modified list

```
/**
* Swaps every two adjacent nodes and returns its head.
* Note: This function assumes the existence of a ListNode type.
* @param {ListNode | null} head - The head of the linked list.
* @return {ListNode | null} - The new head of the modified list.
*/
function swapPairs(head: ListNode | null): ListNode | null {
   // Create a dummy node to simplify edge cases.
    let dummy = createListNode(0, head);
   // Initialize pointers for the previous and current nodes.
    let previous = dummy;
    let current = head;
```

// Point the next node back to the current node, effecting the swap.

# 'prev\_node.next' is adjusted to point to 'temp' after swapping

# Move 'prev\_node' and 'current\_node' forward in the list

prev\_node, current\_node = current\_node, current\_node.next

// Link the previous node to the new head of the swapped pair.

const createListNode = (val: number, next: ListNode | null = null): ListNode => {

```
return dummy.next;
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next_node=None):
        self.val = val
        self.next = next_node
class Solution:
```

def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]: # Create a dummy node that points to the head of the list dummy = ListNode(next\_node=head) prev\_node, current\_node = dummy, head # Iterate through the list while there are pairs to swap while current\_node and current\_node.next: # 'temp' is the node that will be swapped with the current node temp = current\_node.next # Adjust 'current\_node.next' to the node after 'temp' current\_node.next = temp.next # The 'temp' node now should point to 'current\_node' after swapping temp.next = current\_node

The time complexity of the code is O(n) where n is the number of nodes in the given linked list. This complexity arises because the algorithm must visit each node to swap the pairs, traversing the entire length of the list once.

The space complexity of the code is 0(1) because it only uses a constant amount of extra space. The variables dummy, pre, cur, and t are used for manipulation, but the space occupied by these variables does not scale with the size of the input list, thus constituting a constant space complexity.