Solution: Reverse Nodes in k-Group

Let's solve the Reverse Nodes in k-Group problem using the In-Place Reversal of Linked List pattern.

We'll cover the following

Statement

The task is to reverse the nodes in groups of k in a given linked list, where k is a positive integer, and at most the length of the linked list. If any remaining nodes are not part of a group of k, they should remain in their original order.

It is not allowed to change the values of the nodes in the linked list. Only the order of the nodes can be modified.

> **Note:** Use only $O(1)$ extra memory space.

Constraints:

Let `n` be the number of nodes in a linked list.

- $1 \leq$ `k` $\leq$ `n` ≤ 500
- $0 \leq$ `Node.value` ≤ 1000

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let's explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

A naive approach would be to use another data structure—like a stack—to reverse the nodes of the linked list and then create a new linked list with reversed nodes. Here's how the algorithm works:

- We iterate the linked list.
- We push the k group of nodes to the stack.

- We pop all k numbers of nodes from the stack and add the nodes to a new linked list. When we do this, the stack will give us the reversed nodes in the k group.
- We repeat the above steps for every group of size k present in our linked list.
- In the end, if there are less than k nodes left in the original linked list, we'll point the tail of the reversed linked list to the remaining nodes of the original linked list.

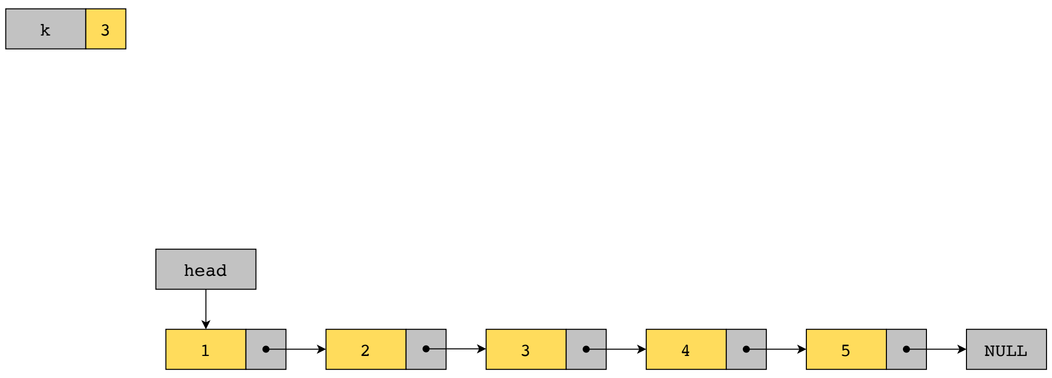
The time complexity of this solution is $O(n)$, since we traverse the linked list once. However, the space complexity is $O(n+k)$, where n is the length of the linked list to store the reversed elements and k is the length of the stack. If a linked list contains thousands of nodes, we need to allocate a lot of memory resources to solve this problem. Let's see if we can use the in-place linked list reversal pattern to reduce the space complexity of our solution.

Optimized approach using in-place reversal of a linked list

The optimized approach is to use less space in memory. We actually need to reverse each group of k nodes in place. We can think of each k-group of nodes as a separate linked list. For each of these linked lists, applying an in-place linked list reversal solves the original problem. We need to invoke the in-place reversal of linked list code $\lceil n/k \rceil$ times, where n is the size of the linked list.

The slides below illustrate how we would like the algorithm to run:

We will try to reverse groups of **k** nodes in below linked list.



1 of 31

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

We will first traverse the linked list and check which groups of k nodes can be reversed. Here is how the algorithm works:

- We initialize a node, `dummy`, and attach it to the start of the linked list, i.e., by setting its `next` pointer equal to the head.
- We set a pointer, `ptr`, equal to the `dummy` node. We will use this pointer to traverse the linked list.
- We traverse the linked list till `ptr` becomes NULL:
    - We initialize a pointer, `tracker`, to `ptr`. This pointer will be used to keep track of the number of nodes in the current group in the linked list.
    - We use a nested loop to try to move `tracker` k nodes forward in the linked list. If `tracker` becomes NULL before moving k nodes forward, the end of the linked list has been reached and the current group can not be traversed, since it contains less than k nodes. Therefore, we break out of the nested loop. Otherwise, the current group contains k nodes and `tracker` will point to the k^{th} node of the current group.
- After the completion of the nested loop, we check if `tracker` points to NULL:
    - If it does, we've reached the end of the linked list. The current group contains less than k nodes and cannot be reversed. Therefore, we break out of the outer loop, and the algorithm ends.
    - If it does not, the current group contains k nodes and can therefore be reversed.

Java

main.java

ReverseLinkedList.java

LinkedListNode.java

LinkedList.java

PrintList.java

```java
class Reorder {

        public static void reverseKGroups(LinkedListNode head, int k) {

        // Create a dummy node and set its next pointer to the head
        LinkedListNode dummy = new LinkedListNode(0);
        dummy.next = head;
        LinkedListNode ptr = dummy;

        while (ptr != null) {
            System.out.println("\tIdentifying a group of " + k + " nodes:");
            System.out.println("\t\tptr: " + ptr.data);

            // Keep track of the current position
            LinkedListNode tracker = ptr;

            System.out.print("\t\tCurrent group: ");

            // Traverse k nodes to check if there are enough nodes to reverse
            for (int i = 0; i < k; i++) {
                if (tracker == null) {
                    break;
                }
                tracker = tracker.next;
                if (tracker != null) {
                    System.out.print(tracker.data + " ");
                } else {
                    System.out.print("");
```

Reverse Nodes in k-Group

Note: The `printListWithForwardArrow` method is used to print the linked list with arrows. It isn't part of the solution logic.

The next step is to reverse the first group of k nodes. Here is how the algorithm works:

- We modify the existing `reverseLinkedList` function in the `LinkedListReversal.java` file to take an additional parameter, `k`, which specifies the number of nodes to reverse.
- In the `reverseLinkedList` function, for the case where `tracker` does not point to NULL, we declare three pointers:
  - `current`
  - `previous`
  - `next`
- We call the `reverseLinkedList` function, which reverses the current group of nodes and updates the above three pointers by returning their values.
- After the reversal, we have a fragmented group that has been separated from the rest of the list. The `previous` pointer now points to the first node of the reversed group while the `current` and `next` pointers now point to the first node of the next group.
- We break out of the outer loop to end the algorithm once the first group has been reversed.

Note: Changes were made in the following two files:

- `main.py`
- `LinkedListReversal.java`

Java

main.java
LinkedListReversal.java
LinkedListNode.java
LinkedList.java
PrintList.java

```java
class LinkedListReversal{
    static LinkedListNode[] reverseLinkedList(LinkedListNode node, int k){

        LinkedListNode previous = null;
        LinkedListNode current = node;
        LinkedListNode next = null;

        for (int i = 0; i < k; i++) {
            // temporarily store the next node
            next = current.next;
            // reverse the current node
            current.next = previous;
            // before we move to the next node, point previous to the
            // current node
            previous = current;
            previous = current;
            // move to the next node
            current = next;
        }

        System.out.println("\t\tPointers after reversing k = " + k + " elemer
        System.out.println("\t\t\tcurrent: " + (current != null ? Integer.toS
        System.out.println("\t\t\tnext: " + (next != null ? Integer.toString(
        System.out.println("\t\t\tprevious: " + (previous != null ? Integer.t

        return new LinkedListNode[]{previous, current};
    }
}
```

Reverse Nodes in k-Group

After reversing the first group of k nodes, we need to reattach it to the rest of the linked list. Here is how the algorithm works:

- We first need to access the last node in the reversed group. The `ptr` pointer is currently pointing to the node immediately before the last node of the reversed group. We initialize a new pointer, `lastNodeOfReversedGroup`, and set it equal to the next node of `ptr`. This node now points to the last node of the reversed group.
- We now need to link the last node of the reversed group to the first node of the linked list coming after it. The `current` pointer is currently pointing to the first node of the next group. We set the next node of `lastNodeOfReversedGroup` to the `current` pointer.
- We now need to link the first node of the reversed group to the last node of the linked list that comes before it. The `previous` node is currently pointing to the first node of the reversed group. We set the next node of `ptr` equal to the `previous` pointer.
- Lastly, we need to set the `ptr` pointer equal to the last node of the reversed group, which resets its position so that we can attempt to reverse the next group. We do this by setting the `ptr` pointer equal to the `lastNodeOfReversedGroup` pointer.
- We break out of the outer loop to end the algorithm once the first reversed group has been reattached to the linked list.

Java

main.java

LinkedListReversal.java

LinkedListNode.java

LinkedList.java

PrintList.java

```
class Reorder {

    public static void reverseKGroups(LinkedListNode head, int k) {

        // Create a dummy node and set its next pointer to the head
        LinkedListNode dummy = new LinkedListNode(0);
        dummy.next = head;
        LinkedListNode ptr = dummy;

        while (ptr != null) {
            System.out.println("\tIdentifying a group of " + k + " nodes:");
```

```java
            System.out.println("\t\tptr: " + ptr.data);

            // Keep track of the current position
            LinkedListNode tracker = ptr;

            System.out.print("\t\tCurrent group: ");

            // Traverse k nodes to check if there are enough nodes to reverse
            for (int i = 0; i < k; i++) {
                if (tracker == null) {
                    break;
                }
                tracker = tracker.next;
                if (tracker != null) {
                    System.out.print(tracker.data + " ");
                } else {
                    System.out.print("");
```

Reverse Nodes in k-Group

Finally, the last step is to repeat the above process for all groups of k nodes. This is done by simply not breaking out of the outer loop once the first group has been reversed and attached. After the linked list has been traversed, i.e., `ptr` becomes NULL, we return the next node of `dummy`, which contains the reversed linked list attached to it.

Java

main.java

LinkedListReversal.java

LinkedListNode.java

LinkedList.java

PrintList.java

```java
class Reorder {

    public static LinkedListNode reverseKGroups(LinkedListNode head, int k) {

        // Create a dummy node and set its next pointer to the head
        LinkedListNode dummy = new LinkedListNode(0);
        dummy.next = head;
        LinkedListNode ptr = dummy;

        while (ptr != null) {
            System.out.println("\tIdentifying a group of " + k + " nodes:");
            System.out.println("\t\tptr: " + ptr.data);

            // Keep track of the current position
            LinkedListNode tracker = ptr;

            System.out.print("\t\tCurrent group: ");

```

```java
                if (tracker == null) {
                    break;
                }
                tracker = tracker.next;
                if (tracker != null) {
                    System.out.print(tracker.data + " ");
                } else {
                    System.out.print("");
```

Reverse Nodes in k-Group

Just the code

Here’s the complete solution to this problem:

Java

main.java

LinkedListReversal.java

LinkedListNode.java

LinkedList.java

PrintList.java

```java
class LinkedListReversal{
    static LinkedListNode[] reverseLinkedList(LinkedListNode node, int k){

        LinkedListNode previous = null;
        LinkedListNode current = node;
        LinkedListNode next = null;

        for (int i = 0; i < k; i++) {
            next = current.next;
            current.next = previous;
            previous = current;
            current = next;
        }

        return new LinkedListNode[]{previous, current};
    }
}
```

Reverse Nodes in k-Group

Solution summary

To recap, the solution to this problem can be divided into the following four main parts:

- Check if there are k nodes present in the current group.
- If the current group contains k nodes, reverse it.
- Reattach the reversed group to the rest of the linked list.
- Repeat the process above until there are less than k nodes left in the linked list.

Time complexity

The time complexity of this solution is $O(n)$, where n is the number of nodes in the list.

Space complexity

The space complexity of this solution is $O(1)$, since we’ll use a constant number of additional variables to maintain the connections between the nodes during reversal.

Back

Reverse Nodes in k-Gr...

Next

Reverse Linked List II

✓ Mark as Completed