Solution Review: Move Tail to Head

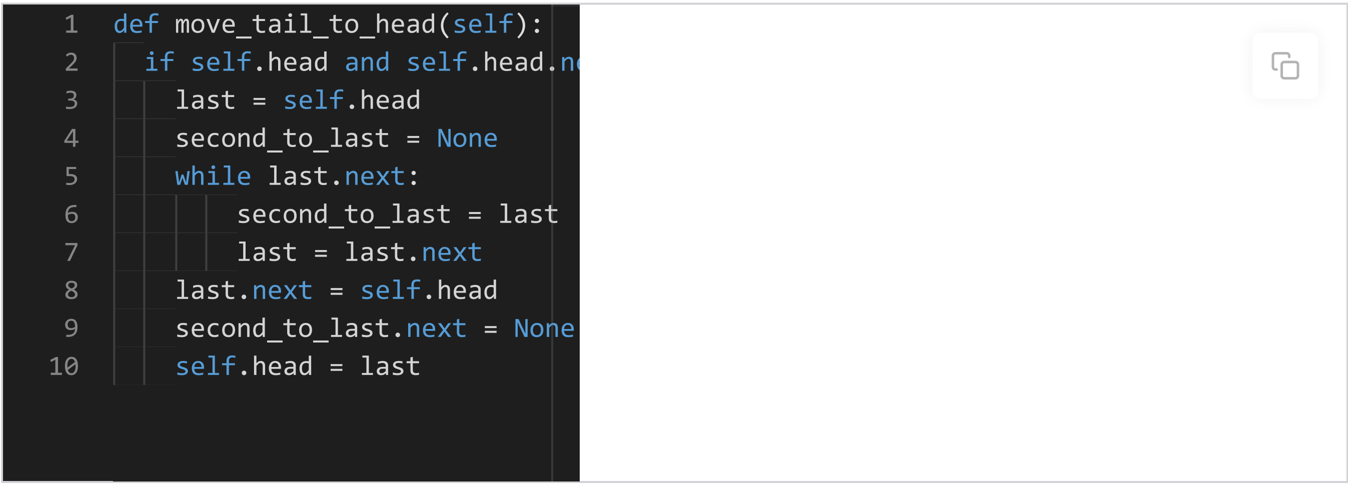
This lesson contains the solution review for the challenge of moving the tail node of a linked list to the head.

We'll cover the following

- Implementation
- Explanation

For this problem, we will use two pointers where one will keep track of the last node of the linked list, and the other will point to the second-to-last node of the linked list. Let's have a look at the code below.

Implementation



```
 1  def move_tail_to_head(self):
 2      if self.head and self.head.ne
 3          last = self.head
 4          second_to_last = None
 5          while last.next:
 6              second_to_last = last
 7              last = last.next
 8          last.next = self.head
 9          second_to_last.next = None
10          self.head = last
```

Explanation

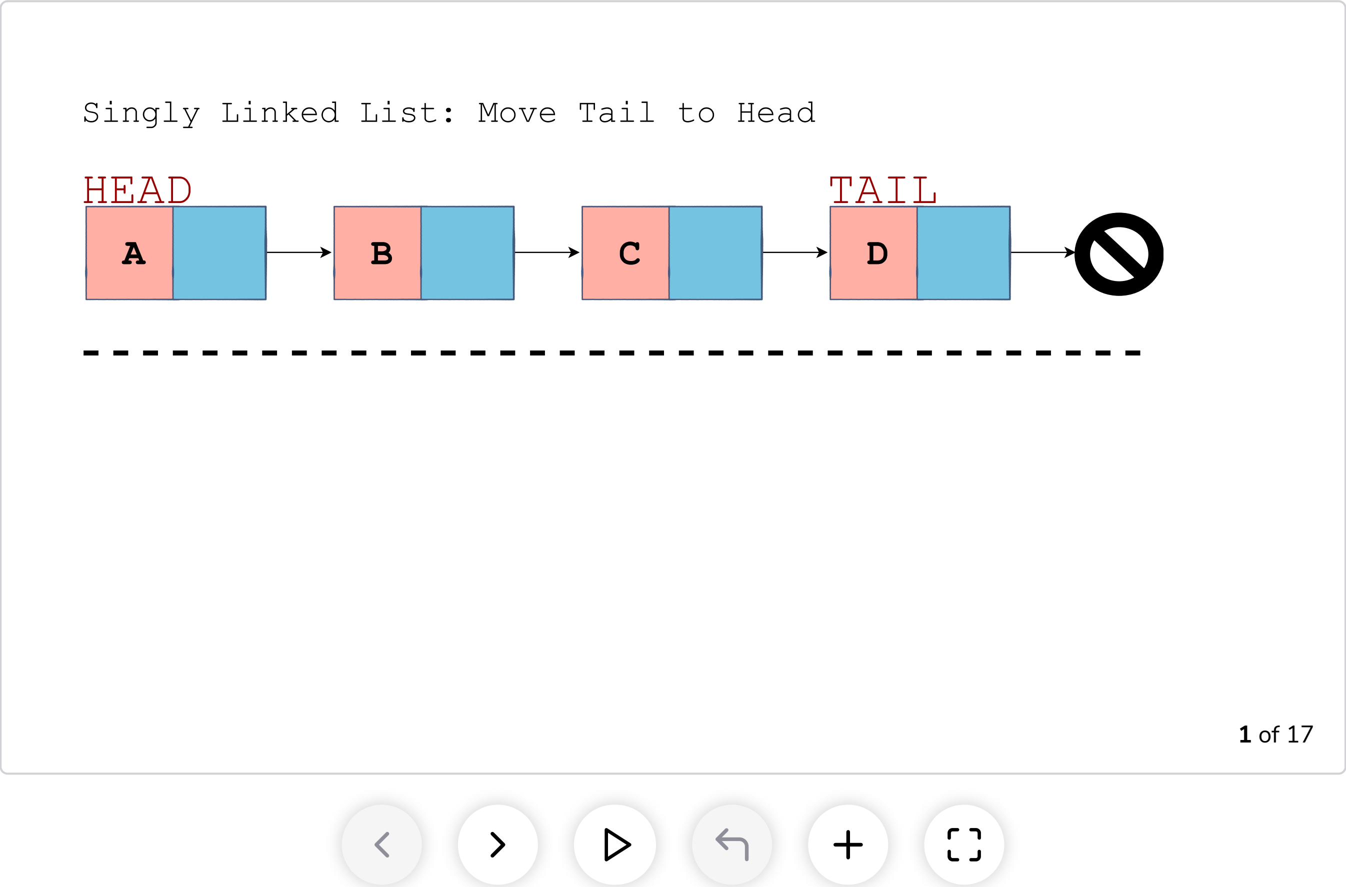
Let's go over a line by line explanation of the solution above.

Line 2 ensures that the code proceeds to **line 3** if there is more than one element in the linked list. This implies `self.head` and `self.head.next` is not `None`. We initialize `last` and `second_to_last` to `self.head` and `None` on **lines 3-4**. Now we have to make them point to what they are supposed to point to, i.e., `last` should point to the last node while `second_to_last` should point to i.e., `last` should point to the last node while `second_to_last` should point to the second to last node in a linked list. Therefore, we traverse the linked list

using the `while` loop on **line 5**. The `while` loop will run until `last.next` becomes `None` which implies that `last` is the last node in the linked list. Before updating `last` to `last.next` on **line 7**, we keep updating `second_to_last` to `last` on **line 6** so that in the last iteration, when `last` is the last node in the linked list, `second_to_last` will be the second to last node in the linked list.

Now that the two pointers are rightly positioned, we have to change a few pointers to complete our solution. Therefore, `last.next` which was previously pointing to `None` is pointed to `self.head` on **line 8**. This makes a circular linked list where the last node points to the first element of the linked list. To make the linked list linear, we make `second_to_last.next` point to `None` on **line 9**. In this way, we have updated the tail pointer of the linked list. At this stage, all that is left is to update the head pointer which we do on **line 10**, and set `self.head` equal to `last`. This completes our solution when we have to move the tail node to the head.

You can visualize the solution above with the help of the slides below:

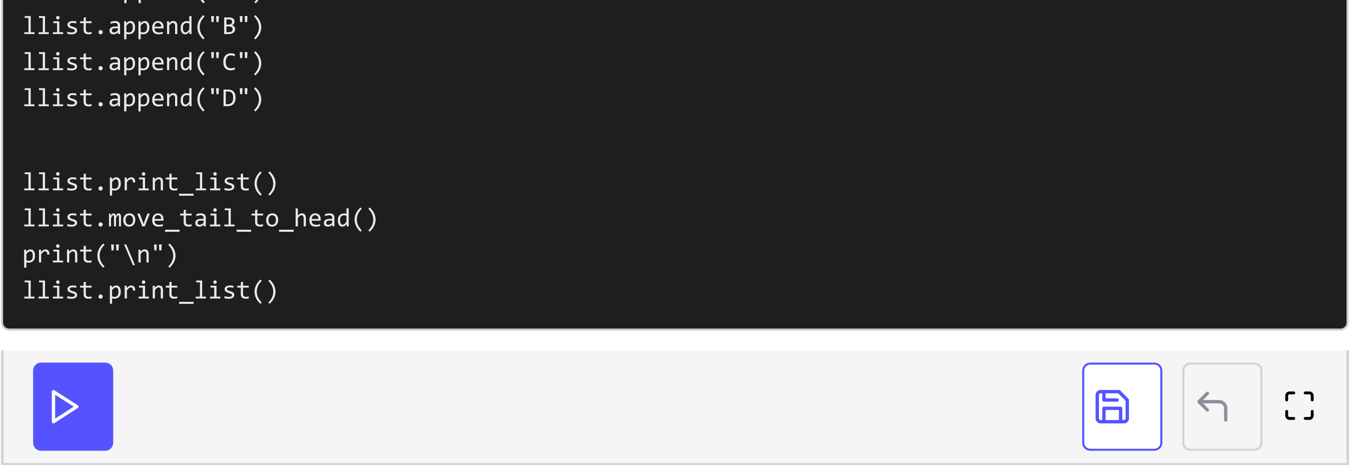


Singly Linked List: Move Tail to Head

I hope the solution was clear to you! Below is the entire implementation of the `LinkedList` class that we have coded up until now. You can play around with

LinkedList class that we have coded up until now. You can play around with and verify the method move_head_to_tail.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None


class LinkedList:
    def __init__(self):
        self.head = None

    def print_list(self):
        cur_node = self.head
        while cur_node:
            print(cur_node.data)
            cur_node = cur_node.next

    def append(self, data):
        new_node = Node(data)

        if self.head is None:
            self.head = new_node
            return

        last_node = self.head
        while last_node.next:
            last_node = last_node.next
        last_node.next = new_node

    def prepend(self, data):
        new_node = Node(data)

        new_node.next = self.head
        self.head = new_node

    def insert_after_node(self, prev_node, data):

        if not prev_node:
            print("Previous node does not exist.")
            return

        new_node = Node(data)

        new_node.next = prev_node.next
        prev_node.next = new_node

    def delete_node(self, key):

        cur_node = self.head

        if cur_node and cur_node.data == key:
            self.head = cur_node.next
            cur_node = None
            return

        prev = None
        while cur_node and cur_node.data != key:
            prev = cur_node
            cur_node = cur_node.next
        if cur_node is None:
            return


        prev.next = cur_node.next
        cur_node = None

    def delete_node_at_pos(self, pos):
        if self.head:
            cur_node = self.head

            if pos == 0:
                self.head = cur_node.next
                cur_node = None
                return

            prev = None
            count = 1
            while cur_node and count != pos:
                prev = cur_node
                cur_node = cur_node.next
                count += 1

            if cur_node is None:
                return

            prev.next = cur_node.next
            cur_node = None

    def len_iterative(self):

        count = 0
        cur_node = self.head

        while cur_node:
            count += 1
            cur_node = cur_node.next
        return count

    def len_recursive(self, node):
        if node is None:
            return 0
        return 1 + self.len_recursive(node.next)

    def swap_nodes(self, key_1, key_2):

        if key_1 == key_2:
            return

        prev_1 = None
        curr_1 = self.head
        while curr_1 and curr_1.data != key_1:
            prev_1 = curr_1
            curr_1 = curr_1.next

        prev_2 = None
        curr_2 = self.head
        while curr_2 and curr_2.data != key_2:
            prev_2 = curr_2
            curr_2 = curr_2.next

        if not curr_1 or not curr_2:
            return

        if prev_1:
            prev_1.next = curr_2
        else:
            self.head = curr_2

        if prev_2:
            prev_2.next = curr_1
        else:
            self.head = curr_1

        curr_1.next, curr_2.next = curr_2.next, curr_1.next

    def print_helper(self, node, name):
        if node is None:
            print(name + ": None")
        else:
            print(name + ":" + node.data)

    def reverse_iterative(self):

        prev = None
        cur = self.head
        while cur:
            nxt = cur.next
            cur.next = prev

            self.print_helper(prev, "PREV")
            self.print_helper(cur, "CUR")
            self.print_helper(nxt, "NXT")
            print("\n")

            prev = cur
            cur = nxt
        self.head = prev

    def reverse_recursive(self):

        def _reverse_recursive(cur, prev):
            if not cur:
                return prev

            nxt = cur.next
            cur.next = prev
            prev = cur
            cur = nxt
            return _reverse_recursive(cur, prev)

        self.head = _reverse_recursive(cur=self.head, prev=None)

    def merge_sorted(self, llist):

        p = self.head
        q = llist.head
        s = None

        if not p:
            return q
        if not q:
            return p
        if p and q:
            if p.data <= q.data:
                s = p
                p = s.next
            else:
                s = q
                q = s.next
            new_head = s
        while p and q:
            if p.data <= q.data:
                s.next = p
                s = p
                p = s.next
            else:
                s.next = q
                s = q
                q = s.next
        if not p:
            s.next = q
        if not q:
            s.next = p
        return new_head

    def remove_duplicates(self):

        cur = self.head
        prev = None

        dup_values = dict()

        while cur:
            if cur.data in dup_values:
                # Remove node:
                prev.next = cur.next
                cur = None
            else:
                # Have not encountered element before.
                dup_values[cur.data] = 1
                prev = cur
            cur = prev.next

    def print_nth_from_last(self, n, method):
        if method == 1:
            #Method 1:
            total_len = self.len_iterative()
            cur = self.head
            while cur:
                if total_len == n:
                   #print(cur.data)
                    return cur.data
                total_len -= 1
                cur = cur.next
            if cur is None:
                return

        elif method == 2:
            # Method 2:
            p = self.head
            q = self.head

            count = 0
            while q:
                count += 1
                if(count>=n):
                    break
                q = q.next

            if not q:
                print(str(n) + " is greater than the number of nodes in list.")
                return

            while p and q.next:
                p = p.next
                q = q.next
            return p.data

    def rotate(self, k):
       if self.head and self.head.next:
            p = self.head
            q = self.head
            prev = None
            count = 0

            while p and count < k:
                prev = p
                p = p.next
                q = q.next
                count += 1
            p = prev
            while q:
                prev = q
                q = q.next
            q = prev

            q.next = self.head
            self.head = p.next
            p.next = None

    def count_occurences_iterative(self, data):
        count = 0
        cur = self.head
        while cur:
            if cur.data == data:
                count += 1
            cur = cur.next
        return count

    def count_occurences_recursive(self, node, data):
        if not node:
            return 0
        if node.data == data:
            return 1 + self.count_occurences_recursive(node.next, data)
        else:
            return self.count_occurences_recursive(node.next, data)

    def is_palindrome_1(self):
        # Solution 1:
        s = ""
        p = self.head
        while p:
            s += p.data
            p = p.next
        return s == s[::-1]
    def is_palindrome_2(self):
        # Solution 2:
        p = self.head
        s = []
        while p:
             s.append(p.data)
             p = p.next
        p = self.head
        while p:
            data = s.pop()
            if p.data != data:
                return False
            p = p.next
        return True

    def is_palindrome_3(self):
        if self.head:
            p = self.head
            q = self.head
            prev = []

            i = 0
            while q:
                prev.append(q)
                q = q.next
                i += 1
            q = prev[i-1]

            count = 1

            while count <= i//2 + 1:
                if prev[-count].data != p.data:
                    return False
                p = p.next
                count += 1
            return True
        else:
            return True

    def is_palindrome(self,method):
        if method == 1:
            return self.is_palindrome_1()
        elif method == 2:
            return self.is_palindrome_2()
        elif method == 3:
            return self.is_palindrome_3()

    def move_tail_to_head(self):
        if self.head and self.head.next:
            last = self.head
            second_to_last = None
            while last.next:
                second_to_last = last
                last = last.next
            last.next = self.head
            second_to_last.next = None
            self.head = last

# A -> B -> C -> D -> Null
# D -> A -> B -> C -> Null
llist = LinkedList()
llist.append("A")
```



```
llist.append("B")
llist.append("C")
llist.append("D")


llist.print_list()
llist.move_tail_to_head()
print("\n")
llist.print_list()
```

Hope you liked this exercise. We have another challenge waiting for you in the next lesson. Best of Luck!