

Solution Review: Sum Two Linked Lists

This lesson contains the solution review for the challenge of summing two linked lists.

WE'LL COVER THE FOLLOWING ^

- Implementation
- Explanation

In this lesson, we investigate how to sum two singly linked lists. Check out the code below, after which, we will do a line by line analysis of it.

Implementation

```
def sum_two_lists(self, llist):
    p = self.head
    q = llist.head

    sum_llist = LinkedList()

    carry = 0
    while p or q:
        if not p:
            i = 0
        else:
            i = p.data
        if not q:
            j = 0
        else:
            j = q.data
        s = i + j + carry
        if s >= 10:
            carry = 1
            remainder = s % 10
            sum_llist.append(remainder)
        else:
            carry = 0
            sum_llist.append(s)
        if p:
            p = p.next
        if q:
            q = q.next
    return sum_llist
```



Explanation

First of all, we initialize `p` and `q` to point to the heads of each of the two linked lists (**lines 2-3**). On **line 5**, we declare `sum_llist` and initialize it to a linked list. The data values of `sum_llist` will represent the final sum at the end of this method. `carry` is initialized to `0` on **line 7** and it will help us in evaluating the sum.

With the help of `p` and `q`, we set up a `while` loop which will run until both `p` and `q` equal `None`. On **lines 9-16**, we have handled the cases if either `p` or `q` equal `None`. If `p` or `q` equal `None`, we set `i` or `j` to `0` accordingly. In the other case where `p` and `q` are not equal to `None`, we use the data values of the node they are pointing to and store the data values in variables `i` and `j`. We are using `i` to represent the current digit picked from the first linked list and `j` to represent the current digit picked from the second one.

Once we get the values in `i` and `j`, we evaluate the sum on **line 17** by adding `i`, `j`, and `carry` and storing the sum in variable `s`. Note that `carry` will be `0` in the first iteration. After calculating the sum, we check if that sum is greater than or equal to `10` on **line 18**. If `s` is greater than or equal to `10`, we set `carry` to `1` (**line 19**) and calculate the `remainder` using the modulus operator on **line 20**. Now we append `remainder` to the final linked list (`sum_llist`) on **line 20**. On the other hand, if `s` is less than `10`, then there is no carry (`carry = 0`), and we append `s` to `sum_llist`. These steps are almost the same as we perform the arithmetic operation of addition. On **line 25-28**, we update `p` and `q` to their next nodes if they are not already `None`.

`sum_llist` is returned from the end of the method and contains the sum of the two linked lists we had at the start.

The `sum_two_lists` has been made part of the `LinkedList` class. You can run it and verify our solution!

```
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 = p.next
        else:
            s = q
            q = q.next
        new_head = s
    while p and q:
        if p.data <= q.data:
            s.next = p
            s = p
            p = p.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
```

```
def sum_two_lists(self, llist):  
    p = self.head  
    q = llist.head
```

```
    sum_llist = LinkedList()
```

```
    carry = 0  
    while p or q:  
        if not p:  
            i = 0  
        else:  
            i = p.data  
        if not q:  
            j = 0  
        else:  
            j = q.data  
        s = i + j + carry
```



```

        if s >= 10:
            carry = 1
            remainder = s % 10
            sum_llist.append(remainder)
        else:
            carry = 0
            sum_llist.append(s)
        if p:
            p = p.next
        if q:
            q = q.next
    sum_llist.print_list()

```

```

# 3 6 5
#  4 2
# -----
#
l1list1 = LinkedList()
l1list1.append(5)
l1list1.append(6)
l1list1.append(3)

l1list2 = LinkedList()
l1list2.append(8)
l1list2.append(4)
l1list2.append(2)

print(365 + 248)
l1list1.sum_two_lists(l1list2)

```



I hope you were able to enjoy and understand this challenge. By now, you'll have a firm grasp on the problems concerning singly linked lists. In the next chapter, we are going to explore another type of linked list: Circular Linked List. See you there!