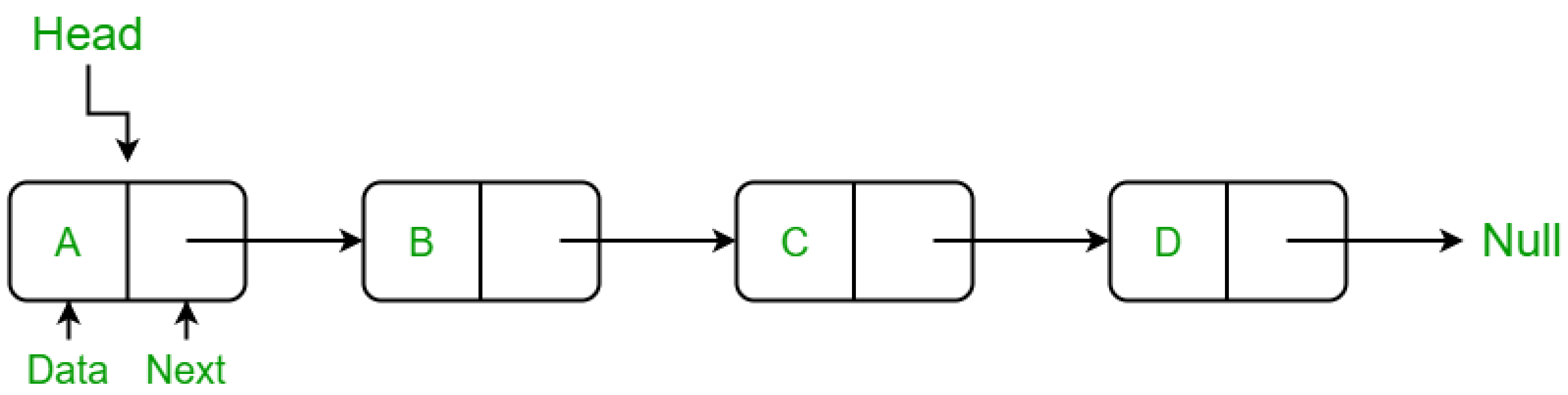
Linked Lists

LinkedIn: Md. Ziaul Karim
Find me here:
Topics: #linked-lists #algorithms #problem-solving #python

What is Linked List in Python?

A linked list is a type of linear data structure similar to arrays. It is a collection of nodes that are linked with each other. A node contains two things first is data and second is a link that connects it with another node. Below is an example of a linked list with four nodes and each node contains character data and a link to another node. Our first node is where **head** points and we can access all the elements of the linked list using the **head.**

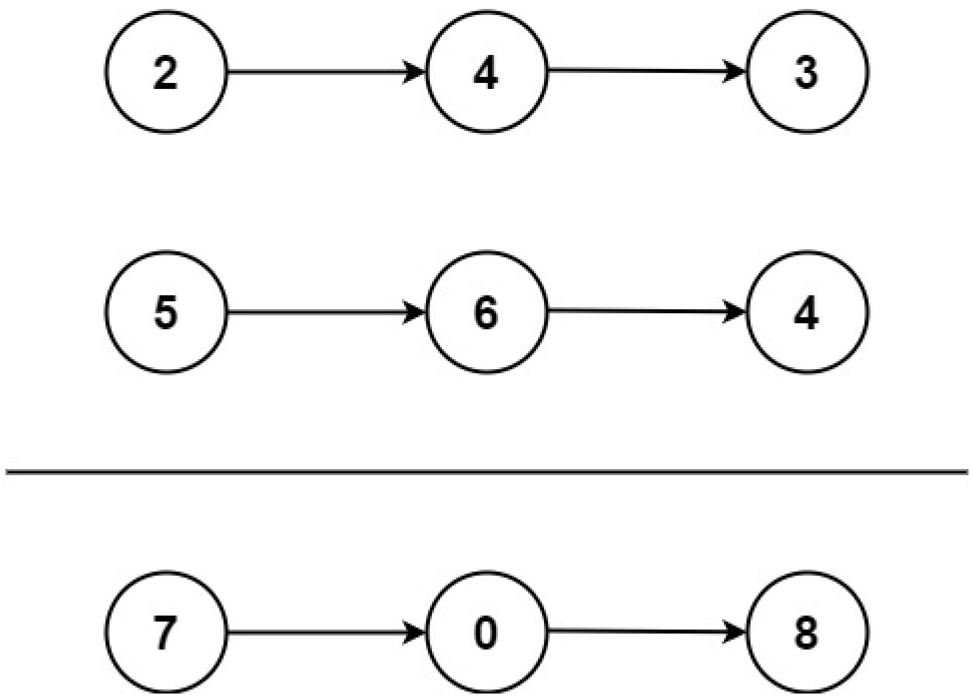


2. Add Two Numbers - Leetcode Problem

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.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:



Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: 342 + 465 = 807.

Example 2:

Input: l1 = [0], l2 = [0]
Output: [0]

Example 3:

Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]

Constraints:

- The number of nodes in each linked list is in the range `[1, 100]`.
- `0 <= Node.val <= 9`
- It is guaranteed that the list represents a number that does not have leading zeros.

Solution

```
class Solution(object):
    def addTwoNumbers(self, l1, l2):
        """
        :type l1: ListNode
        :type l2: ListNode
        :rtype: ListNode
        """
        res=[]
        keep=0
        while l1 and l2:
            # First we check the constraints
            if l1.next == None and l2.next: # if we ran out of l1, but we still have elements in l2
                l1.next = ListNode() # we expand l1 to match l2
            elif l1.next and l2.next==None: # similarly if we ran out of l2, but we still have elements in l1
                l2.next = ListNode() # we expand l2 to match l1
            add = l1.val + l2.val + keep
            if add >= 10: #if our sum amounts to 10 or greater
                res.append(add-10) # maximum sum of two single digit numbers is 18, we append 8
                keep = 1 # and keep 1
            else: # if our sum doesn't exceed or equal to 10
                res.append(add) # then we add it as it is and
                keep = 0 # reset our keep to zero
            l1, l2 = l1.next, l2.next # then we move to the next elements in both of our Linked List
        if keep: res.append(keep) # after all the nodes are traverse and we still have a value in our keep,we
append it to our result
        return ListNode._array_to_list_node(res) #then convert it to a ListNode and return
```

Input

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

Output

```
[7,0,8]
```

More about linked list (under the hood)

```python
# courtesy for this code goes to © geeksforgeeks.com
# Explained by © Md. Ziaul Karim
# This is our Node class that creates an instance of a new element in the list
class Node:
    #this is our constructor, that takes data as an argument, puts it in the data object & initializs the next pointer as None by default
    def __init__(self,data):
        self.data = data # Node
        self.next = None # Pointer to the next

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

    def insertAtBegin(self, data:int) -> None:
        new_node = Node(data) # Initializes the data as an instance of the Node class, which has two objects, the data itself and a pointer to the next, which is None by default

        """
        This is an insert operation at the beginning of a linked list.
        You might as well know in python insert operation, it comes as something like this => some_list.insert(index, element)
        Now we know that, The beginning element of a linked list, would point to a HEAD, which marks it as the first element,
        with this head we can access all the elements of a linked list.
        This is just a way of pointing out which direction marks the beginning of your list (if it isn't obvious ofcourse)
        For an example, if we have a list [3,2,7] and you are told that the digits are stored in reversed order,
        that means the HEAD of the linked list is pointed at the last element, in this case index 2
        Here's a sneakpeek
        data = 3, next = None
        data = 2, next = 3
        data = 7, next = 2 -> HEAD
        list = Node{
            val:7,
            next: Node{
                val: 2,
                next: Node{
                    val: 3,
                    next: None
                }
            }
        }
        """
        # if there is no marked HEAD already, then it could only mean that we are inserting the first element into an empty linked list.
        if self.head is None:
            # if it is the first element we are inserting then surely we have to mark it as the HEAD
            self.head = new_node
            # then we end the operation here
            return

        #this is in case of insert operation to a non-empty list
        else:
            #if HEAD is not found to be empty, then old head becomes the new next
            new_node.next = self.head
             # new node becomes the new head
            self.head = new_node
```

```python
    # Method to add a node at the end of LL
    def insertAtEnd(self, data):
        new_node = Node(data)
        #if there is no HEAD, that means we are inserting in an empty list
        if self.head is None:
            #new node becomes the HEAD
            self.head = new_node
            return
        # else we have to traverse the whole list, beginning from the HEAD
        current_node = self.head
        # while current node still has some element next to it
        while(current_node.next):
            # we go to the next node
            current_node = current_node.next
        #once we have reached to a place where the current_node doesn't have anything to it's next, meaning current_node.next = None
        #we set it's next element to be our new_node
        current_node.next = new_node

    """
    Now we move on to the next part, which means, we are going to run insert operation at predetermined positions.
    Kind of like the some_list(index, element)
    It good to note that indexing always starts at position 0, hence we set position = 0.
    The
    """
    def insertAtIndex(self, data, index:int) -> None:
            new_node = Node(data) # Initializes the data as an instance of the Node class, which has two objects, the data itself and a pointer to the next, which is None by default
            current_node = self.head #We say, our current_node, sits at the HEAD.
            position = 0 # by default we say, the position is 0
            # if index is 0, then we insert at the beginning
            if position == index:
                self.insertAtBegin(data)
            #otherwise
            else:
                # we traverse across the list, as long as our current_node is occupied and the position doesn't match our determined index
                while(current_node != None and position+1 != index):
                    position = position+1 # new_node: "What? This ain't the right address? A'ight, I'm going next_door. 🚶 "
                    current_node = current_node.next #current_node: gets updated to next door address.

                # Once that is done, we might or might not have arrived at the previous index/address of our desired index /address, which is at our current_node

                # current_node to new_node -> "Oh yeah, the address you're looking for is just next door."
                if current_node != None:
                    # Then our new_node proceeds evict the guy next door to it's next door. Squeezing in.
                    new_node.next = current_node.next
                    # Then the current node welcomes the new_node as it's new neighbor. - "Welcome bro! you move in to my next door.""
                    current_node.next = new_node # "What up brother! Let me just squeeze in here."

                # This part is when your new_node couldn't find a place to go in. Means, it ran out of addresses and current_node = None
                else:
                    #"Looks like this ain't gonna work brother!" -> Linked list said calmly to the new_node
                    print("Index not present")
                    # new_node: "Kay! I'mma yeet myself outta here. Laters Gators! 🫡 "


    # Update node of a linked list
```

```python
    # at given position
    def updateNode(self, val, index:int) -> None:
        current_node = self.head
        position = 0
        # if index is 0, then we update the head
        if position == index:
            #replaces the data at current_node with given val
            current_node.data = val
        #otherwise
        else:
            # we traverse across the list, as long as our current_node is occupied and the position doesn't match our determined index
            while(current_node != None and position != index):
                position = position+1 # val: "What? This ain't the right address? A'ight, I'm going next_door. 🚶"
                current_node = current_node.next #current_node: gets updated to next door address.

            # Once that is done, we might or might not have arrived at the previous index/address of our desired index /address, which is at our current_node

            # current_node to val -> "Oh yeah, this is the address. I need an update!"
            if current_node != None:
                current_node.data = val #val: Kay! Never mind me.

            # This part is when our val didn't find a place to go in. Means, it ran out of addresses and current_node = None
            else:
                print("Index not present")
                # val: "Kay! I'mma yeet myself outta here. Laters Gators! 🫡"

    #method to remove first node from the list
    #the method to remove first node should be considered as a separate operation
    # It's pretty simple, we just move the head to it's next position
    def remove_first_node(self):
        # if head doesn't exit, then the list is empty, so there's nothing to remove
        if(self.head == None):
            return
        # The next element is the new HEAD
        self.head = self.head.next

    # Method to remove last node of linked list
    def remove_last_node(self):
        # if head doesn't exit, then the list is empty, so there's nothing to remove
        if self.head is None:
            return
        # Otherwise,
        # We traverse the list till we reach the last node that has next = None
        current_node = self.head
        try:
            # we keep going next till we reach [..........current_node, next, next]
            while(current_node.next.next):
                current_node = current_node.next # now we are here -> [..........current_node, next]
        except:
            return
        # Now we say [.........., current_node], breaking away the connection with the last node
        print(f"Removing -> '{current_node.next.data}'")
        current_node.next = None

    # Method to remove at given index
    def remove_at_index(self, index):
        # if head doesn't exit, then the list is empty, so there's nothing to remove
        if self.head == None:
            return
        # Otherwise,
```

```
        current_node = self.head
        position = 0
        #if index is 0
        if position == index:
            self.remove_first_node() # remove the first node
        #otherwise
        else:
            # We traverse the list starting at position 0, as long as our current_node is occupied and the position doesn't match our determined index,
            # if it matches we break the loop,
            # even if it doesn't we complete the loop, and end up at the last index, where current_node's next is None
            while(current_node != None and position+1 != index):
                position = position+1  # index: "What? This ain't the right address? A'ight, I'm going next_door. 🚶 "
                current_node = current_node.next #current_node: gets updated to next door address.

            # Once that is done, we might or might not have arrived at the previous index/address of our desired index /address, which is at our current_node

            # current_node: "Oh yeah, the address you're looking for is just next door." -> [...., current_node, target_index, next, next, ....]
            if current_node != None:
                try:
                    # We place the address next to the target_index to the next address
                    print(f"Removing -> '{current_node.next.data}'")
                    current_node.next = current_node.next.next  # -> [current_node, next, next, ....]
                except:
                    return
            else:
                print("Index not present")

    # Method to remove a node from linked list
    def remove_node(self, data):
        current_node = self.head
        #Traverse the list, till the data matches to the next of the current_node, basically [...., current_node, target_node, next, next, ....]
        while(current_node != None and current_node.next.data != data):
            current_node = current_node.next
        # if it reaches the end of the list without finding a match, then we return
        if current_node == None:
            return
        # if it finds a match -> [...., current_node, target_node, next, next, ....]
        else:
            print(f"Removing -> '{current_node.next.data}'")
            current_node.next = current_node.next.next # then -> [...., current_node, next, next, ....]

    # Print the size of linked list
    def sizeOfLL(self):
        size = 0
        if(self.head):
            current_node = self.head
            # while a current_node exists
            while(current_node):
                size = size+1 # we add it to the size
                current_node = current_node.next # then we move to the next
            return size # then we return size
        else:
            # in case the list is empty or the HEAD wasn't found
            return 0

    # print method for the linked list
    def printLL(self):
        # Starting from the HEAD
```

```
        current_node = self.head
        linked_list=[] # this is the list where we store every element to show it as a normal list
        while(current_node): # while current_node exists
            linked_list.append(current_node.data) # append the data of the current node to the empty list
            current_node = current_node.next # move to the next node
        print(linked_list) #print the linked list

if __name__ == "__main__":
    # create a new linked list
    llist = LinkedList()

    # add nodes to the linked list
    llist.insertAtEnd('a')
    llist.insertAtEnd('b')
    llist.insertAtBegin('c')
    llist.insertAtEnd('d')
    llist.insertAtIndex('g', 2)

    # print the linked list
    print("Node Data")
    llist.printLL()
    print("\nHEAD of linked list:", end =" ")
    print(llist.head.data)
    # remove a nodes from the linked list
    print("\nRemove First Node")
    llist.remove_first_node()
    llist.printLL()
    print("\nHEAD of linked list:", end =" ")
    print(llist.head.data)
    print("\nRemove Last Node")
    llist.remove_last_node()
    llist.printLL()
    print("\nRemove Node at Index 1")
    llist.remove_at_index(1)
    llist.printLL()


    # print the linked list again
    print("\nLinked list after removing a node:")
    llist.printLL()

    print("\nUpdate node Value")
    llist.updateNode('z', 0)
    llist.printLL()

    print("\nSize of linked list :", end=" ")
    print(llist.sizeOfLL())

    print("\nHEAD of linked list:", end =" ")
    print(llist.head.data)
```

Output

```
Node Data
['c', 'a', 'g', 'b', 'd']

HEAD of linked list: c

Remove First Node
['a', 'g', 'b', 'd']
```

```
HEAD of linked list: a

Remove Last Node
Removing -> 'd'
['a', 'g', 'b']

Remove Node at Index 1
Removing -> 'g'
['a', 'b']

Linked list after removing a node:
['a', 'b']

Update node Value
['z', 'b']

Size of linked list : 2

HEAD of linked list: z
```