Leetcode 简单题(python版) 链表部分

21合并两个有序链表

题目：

将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

**示例：**

**输入：**1->2->4, 1->3->4

**输出：**1->1->2->3->4->4

思路：首先要确定代码的鲁棒性，考虑几个特殊情况，然后依次的进行大小比较即可

代码：

1. # Definition for singly-linked list.
2. # class ListNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution:
8. **def** mergeTwoLists(self, l1, l2):
9. """
10. :type l1: ListNode
11. :type l2: ListNode
12. :rtype: ListNode
13. """
14. **if** **not** l1 **and** **not** l2:
15. **return** None
16. **if** **not** l1:
17. **return** l2
18. **if** **not** l2:
19. **return** l1
20. first=ListNode(-1)
21. start=first
22. **while** l1 **and** l2:
23. **if** l1.val<=l2.val:
24. start.next=l1
25. start=start.next
26. l1=l1.next
27. **else**:
28. start.next=l2
29. start=start.next
30. l2=l2.next
31. **if** l1:
32. start.next=l1
33. **if** l2:
34. start.next=l2
35. **return** first.next

83删除排序链表中的重复元素

题目：给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。

思路：直接法，要注意到最后return 的是head,而不是current

代码：

1. # Definition for singly-linked list.
2. # class ListNode:
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution:
8. **def** deleteDuplicates(self, head):
9. """
10. :type head: ListNode
11. :rtype: ListNode
12. """
13. **if** head **is** None:
14. **return** None
15. current=head
16. **while** current.next:
17. **if** current.val==current.next.val:
18. current.next=current.next.next
19. **else**:
20. current=current.next
21. **return** head

141环形链表

题目：给定一个链表，判断链表中是否有环

思路：利用快慢指针，如果快指针会和慢指针重合，那就肯定有环，如果快指针的next==None，则说明无环

代码：

1. # Definition for singly-linked list.
2. # class ListNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution(object):
8. **def** hasCycle(self, head):
9. """
10. :type head: ListNode
11. :rtype: bool
12. """
13. **if** **not** head:
14. **return** False
15. **if** head.next==None:
16. **return** False
17. slow=head
18. fast=head.next
19. **while** slow!=fast:
20. **if** fast==None **or** fast.next==None:
21. **return** False
22. slow=slow.next
23. fast=fast.next.next
24. **return** True

160相交链表

题目：编写一个程序，找到两个单链表相交的起始节点。

思路：

思路是这样的（题目中假设没有环）：   
1.分别遍历两个链表，如果尾节点不同则不相交，返回None，如果尾节点相同则求相交结点。   
2.求相交结点的方法是，求出链表长度的差值，长链表的指针先想后移动lenA-lenB。然后两个链表一起往后走，若结点相同则第一个相交点。

代码：

1. # Definition for singly-linked list.
2. # class ListNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution(object):
8. **def** getIntersectionNode(self, headA, headB):
9. """
10. :type head1, head1: ListNode
11. :rtype: ListNode
12. """
13. p1=headA
14. p2=headB
15. len1,len2=0,0
16. **while** p1:
17. p1=p1.next
18. len1+=1
19. **while** p2:
20. p2=p2.next
21. len2+=1
22. p1=headA
23. p2=headB
24. **if** len1>len2:
25. **for** i **in** range(len1-len2):
26. p1=p1.next
27. **else**:
28. **for** i **in** range(len2-len1):
29. p2=p2.next
30. **while** p1!=p2:
31. p1=p1.next
32. p2=p2.next
33. **return** p1

203移除链表元素

题目：删除链表中等于给定值 val 的所有节点。

**示例:**

**输入:** 1->2->6->3->4->5->6, ***val*** = 6

**输出:** 1->2->3->4->5

思路：首先要考虑要删除的节点是头节点和尾节点的特殊情况，然后中间节点则统一用一个

代码：

1. # Definition for singly-linked list.
2. # class ListNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution(object):
8. **def** removeElements(self, head, val):
9. """
10. :type head: ListNode
11. :type val: int
12. :rtype: ListNode
13. """
14. **if** **not** head:
15. **return** head
16. pre\_node=ListNode(None)
17. pre\_node.next=head
18. p=pre\_node
19. **while** p.next:
20. **if** p.next.val==val:
21. p.next=p.next.next
22. **else**:
23. p=p.next
24. **return** pre\_node.next

206反转链表

题目：反转一个单链表。

**示例:**

**输入:** 1->2->3->4->5->NULL

**输出:** 5->4->3->2->1->NULL

思路一：看输入输出，可以存入列表中，反转列表，再依此创建一个新的链表

代码：

1. # Definition for singly-linked list.
2. # class ListNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution(object):
8. **def** reverseList(self, head):
9. """
10. :type head: ListNode
11. :rtype: ListNode
12. """
13. **if** head **is** None **or** head.next **is** None:
14. **return** head
15. p = head
16. d = {}
17. i = 0
18. **while** p:
19. d[i] = p
20. p = p.next
21. i += 1
22. l = len(d)
23. **for** i **in** range(l-1,0,-1):
24. d[i].next = d[i-1]
25. d[0].next = None
26. **return** d[l-1]

思路二：对于每个结点都做反转处理，但是有点难理解，最好画图帮助理解一下

代码：

1. **if** head **is** None:
2. **return** None
3. cur = head
4. pre = None
5. nxt = cur.next
6. **while** nxt:
7. cur.next = pre
8. pre = cur
9. cur = nxt
10. nxt = nxt.next
11. cur.next = pre
12. head = cur
13. **return** head

234回文链表

题目：请判断一个链表是否为回文链表。

进阶：你能否用 O(n) 时间复杂度和 O(1) 空间复杂度解决此题？

**示例 1:**

**输入:** 1->2

**输出:** false

**示例 2:**

**输入:** 1->2->2->1

**输出:** true

思路1：遍历整个链表，将链表每个节点的值记录在数组中，再判断数组是不是一个回文数组，时间复杂度为O（n），但空间复杂度也为O（n），不满足空间复杂度要求。

代码：

1. **class** Solution:
2. **def** isPalindrome(self, head):
3. """
4. :type head: ListNode
5. :rtype: bool
6. """
7. **if** head **is** None **or** head.next **is** None:
8. **return** True
9. l = []
10. p = head
11. **while** p.next:
12. l.append(p.val)
13. p = p.next
14. l.append(p.val)
15. **return** l == l[::-1]

思路2：利用栈先进后出的性质，将链表前半段压入栈中，再逐个弹出与链表后半段比较。时间复杂度O（n），但仍然需要n/2的栈空间，空间复杂度为O（n）

思路3：可以设置快慢指针，当快指针走完时，慢指针刚好走到中点，随即原地将后半段反转。然后进行回文判断，符合进阶要求

代码：

1. # Definition for singly-linked list.
2. # class ListNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution(object):
8. **def** isPalindrome(self, head):
9. """
10. :type head: ListNode
11. :rtype: bool
12. """
13. **if** head **is** None **or** head.next **is** None:
14. **return** True
15. **if** head.next.next **is** None:
16. **return** head.val == head.next.val
17. fast = slow = q = head
18. **while** fast.next **and** fast.next.next:#这里快指针的判读条件跟判断环形有一点不同
19. fast = fast.next.next
20. slow = slow.next
21. **def** reverse\_list(head):
22. **if** head **is** None:
23. **return** head
24. cur = head
25. pre = None
26. nxt = cur.next
27. **while** nxt:
28. cur.next = pre
29. pre = cur
30. cur = nxt
31. nxt = nxt.next
32. cur.next = pre
33. **return** cur
34. p = reverse\_list(slow.next)
35. **while** p.next:
36. **if** p.val != q.val:
37. **return** False
38. p = p.next
39. q = q.next
40. **return** p.val == q.val

237删除链表中的节点

题目：请编写一个函数，使其可以删除某个链表中给定的（非末尾）节点，你将只被给定要求被删除的节点。

现有一个链表 -- head = [4,5,1,9]，它可以表示为:

4 -> 5 -> 1 -> 9

**示例 1:**

**输入:** head = [4,5,1,9], node = 5

**输出:** [4,1,9]

**解释:** 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9.

**示例 2:**

**输入:** head = [4,5,1,9], node = 1

**输出:** [4,5,9]

**解释:** 给定你链表中值为 1 的第三个节点，那么在调用了你的函数之后，该链表应变为 4 -> 5 -> 9.

**说明:**

* 链表至少包含两个节点。
* 链表中所有节点的值都是唯一的。
* 给定的节点为非末尾节点并且一定是链表中的一个有效节点。
* 不要从你的函数中返回任何结果。

思路：太简单了

我们需要知道要删除节点前一个节点，再将上一个节点的next指针指向删除节点后的指针。但这道题是单向链表，没法知道前一个节点，所以需要一些奇技淫巧。   
因为题目给的是删除节点，那说明这个节点可以舍弃了，我们把下一个节点的值拷贝给当前要删除的节点，再删除下一个节点

代码：

1. # Definition for singly-linked list.
2. # class ListNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution(object):
8. **def** deleteNode(self, node):
9. """
10. :type node: ListNode
11. :rtype: void Do not return anything, modify node in-place instead.
12. """
13. node.val = node.next.val
14. node.next = node.next.next

707设计链表

题目：

设计链表的实现。您可以选择使用单链表或双链表。单链表中的节点应该具有两个属性：val 和 next。val 是当前节点的值，next 是指向下一个节点的指针/引用。如果要使用双向链表，则还需要一个属性 prev 以指示链表中的上一个节点。假设链表中的所有节点都是 0-index 的。

在链表类中实现这些功能：

* get(index)：获取链表中第 index 个节点的值。如果索引无效，则返回-1。
* addAtHead(val)：在链表的第一个元素之前添加一个值为 val 的节点。插入后，新节点将成为链表的第一个节点。
* addAtTail(val)：将值为 val 的节点追加到链表的最后一个元素。
* addAtIndex(index,val)：在链表中的第 index 个节点之前添加值为 val  的节点。如果 index 等于链表的长度，则该节点将附加到链表的末尾。如果 index 大于链表长度，则不会插入节点。
* deleteAtIndex(index)：如果索引 index 有效，则删除链表中的第 index 个节点。

**提示：**

* 所有值都在 [1, 1000] 之内。
* 操作次数将在  [1, 1000] 之内。
* 请不要使用内置的 LinkedList 库。

**示例：**

思路：使用了结点的方式实现链表，速度比较慢

代码：

1. **class** MyLinkedList(object):
2. **class** Node:
3. **def** \_\_init\_\_(self,data,nextnode):
4. self.val=data
5. self.next=nextnode
7. **def** \_\_init\_\_(self):
8. """
9. Initialize your data structure here.
10. """
11. self.head=None
12. self.count=0
14. **def** get(self, index):
15. """
16. Get the value of the index-th node in the linked list. If the index is invalid, return -1.
17. :type index: int
18. :rtype: int
19. """
20. p=self.head
21. **if** self.count==0 **or** index >self.count-1:
22. **return** -1
23. **else**:
24. i=1
25. **while** i<=index:
26. p=p.next
27. i+=1
28. **return** p.val

31. **def** addAtHead(self, val):
32. """
33. Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.
34. :type val: int
35. :rtype: void
36. """
37. node=self.Node(val,self.head)
38. self.head=node
39. self.count+=1


43. **def** addAtTail(self, val):
44. """
45. Append a node of value val to the last element of the linked list.
46. :type val: int
47. :rtype: void
48. """
49. node=self.Node(val,None)
50. **if** self.count==0:
51. self.head=node
52. self.count+=1
53. **else**:
54. p=self.head
55. **while** p.next!=None:
56. p=p.next
57. p.next=node
58. self.count+=1
60. **def** addAtIndex(self, index, val):
61. """
62. Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted.
63. :type index: int
64. :type val: int
65. :rtype: void
66. """
67. p=self.head
68. post=self.head
69. **if** index==self.count:
70. self.addAtTail(val)
71. **elif** index>self.count:
72. **return** None
73. **else**:
74. i=1
75. **while** i<=index:
76. post=p
77. p=p.next
78. i+=1
79. post.next=self.Node(val,p)
80. self.count+=1

83. **def** deleteAtIndex(self, index):
84. """
85. Delete the index-th node in the linked list, if the index is valid.
86. :type index: int
87. :rtype: void
88. """
89. p=self.head
90. post=self.head
91. **if** self.count==0 **or** index>(self.count-1) **or** index<0:
92. **return** None
93. **else**:
94. i=1
95. **while** i<=index:
96. post=p
97. p=p.next
98. i+=1
99. post.next=p.next
100. self.count-=1

103. # Your MyLinkedList object will be instantiated and called as such:
104. # obj = MyLinkedList()
105. # param\_1 = obj.get(index)
106. # obj.addAtHead(val)
107. # obj.addAtTail(val)
108. # obj.addAtIndex(index,val)
109. # obj.deleteAtIndex(index)

876链表的中间结点

题目：给定一个带有头结点 head 的非空单链表，返回链表的中间结点。

如果有两个中间结点，则返回第二个中间结点。

**示例 1：**

**输入：**[1,2,3,4,5]

**输出：**此列表中的结点 3 (序列化形式：[3,4,5])

返回的结点值为 3 。 (测评系统对该结点序列化表述是 [3,4,5])。

注意，我们返回了一个 ListNode 类型的对象 ans，这样：

ans.val = 3, ans.next.val = 4, ans.next.next.val = 5, 以及 ans.next.next.next = NULL.

**示例 2：**

**输入：**[1,2,3,4,5,6]

**输出：**此列表中的结点 4 (序列化形式：[4,5,6])

由于该列表有两个中间结点，值分别为 3 和 4，我们返回第二个结点。

思路一：当用慢指针**slow**遍历列表时，让另一个指针**fast**的速度是**slow**的二倍，则当快指针到结尾时，**slow**指针位于中间。初始位置都为**head**时，当**fast**指向最终的**null**时，**slow**也就达到了要求

代码：

1. # Definition for singly-linked list.
2. # class ListNode(object):
3. #     def \_\_init\_\_(self, x):
4. #         self.val = x
5. #         self.next = None
7. **class** Solution(object):
8. **def** middleNode(self, head):
9. """
10. :type head: ListNode
11. :rtype: ListNode
12. """
13. slow=head
14. fast=head
15. **while** fast!=None **and** fast.next!=None:
16. slow=slow.next
17. fast=fast.next.next
18. **return** slow