leetcode解题报告-链表

作者：Zing
原文链接：https://ld246.com/article/1456389277130
来源网站：链滴

#前言

##链表的特点

链表的特点是只能从前往后进行顺序访问（当然还有双向链表和环，但到目前为止还未出现过双向链），但是和可以随机访问的数组或列表对比起来，链表在插入，删除时不需要对后续节点进行操作，时间复杂度大大降低。认清楚了链表的特点之后，我们就可以在特定的使用环境下，克服缺点发扬优。

##链表题的特点

应该来说，链表题是广大刷题者又爱又恨的类型。爱是因为，链表题通常不会让人想破脑袋，leetcod中的链表题都是简单的单链表（无双向链表），操作无是增删改查，排序，合并等基础问题。恨是因，链表题通常很琐碎，需要考虑各种特殊情况和越界问题，代码不可避免地会比较长。

根据链表和链表题的特点我们通常也会有一些解题的小技巧：

- 把顺序访问变成随机访问。

这也是我最喜欢用的一种方法，把节点放到列表中。对于python，这实现起来非常方便，python的表可以丢任何类型。

- 很难修改节点，那就修改节点的值。

这是我们很容易忽略的解法。修改节点（如删除或者改变节点顺序），若条件不足（如不知道前驱节），我们可以尝试修改节点的值而不是节点本身。但需要注意题目是否禁止这样的做法。

- 适当增加头指针，尾指针。

增加头指针和尾指针可以很大程度减少我们的特殊条件，使代码行数减少，而且这也是一种好的习惯。

以下对leetcode的链表题进行汇总，基本都是常规的解法和思路。

#删除节点

##19.Remove Nth Node From End of List

- 难度：Easy
- 题目大意：

给定一个链表，删除从末端算起的第N个节点，并返回头节点。假定N总是合理的。

- 思路：

最简单的思路是先扫一遍，获得链表的长度，这样第二次扫到需要删除的节点直接删除就可以。但是样要遍历两次链表。

还有一种解法只需要扫一遍，用三个指针pre,cur,future分别进行记录，pre是cur的前序（删除节点要记录前序），future是探针，保持futrue比cur领先N个节点，当future指向尾节点时，cur就是欲除节点。

这里，为了使当第N个节点为第一个结点时需要做特殊处理，特意加入了一个头结点。这也是链表题用的方法。

- 代码：

```
# Definition for singly-linked list.
class ListNode:
```

```
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    # @param {ListNode} head
    # @param {integer} n
    # @return {ListNode}
    def removeNthFromEnd(self, head, n):
        pre = ListNode(0)
        pre.next = head
        cur = head
        head = pre
        while cur != None:
            future = cur
            for i in range(n-1)
                future = future.next
            if future.next == None:
                pre.next = cur.next
                return head.next
            else:
                pre = cur
                cur = cur.next
```

##83. Remove Duplicates from Sorted List

- 难度：Easy
- 题目大意：

给定一个已经排序的链表，删除所有值重复的节点。

- 思路：

常规的删除节点题目，比较下一节点是否和当前节点值一致，若是删除下一节点，若否向前推进。

- 代码：

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def deleteDuplicates(self, head):
        if not head:
            return head
        c = head
        while c.next:
            n = c.next
            if n.val == c.val:
                c.next = n.next
```

```
        else:
            c = c.next
    return head
```

##82. Remove Duplicates from Sorted List II

- 难度：Medium
- 题目大意：

给定一个已排序链表，删除所有重复出现的节点，即若出现多个1，则删除所有1的节点，而非留下一1。

- 思路：

这道题的难度明显就要比上一题的高出许多。我采取的思路是，一个变量dup记录当前正重复出现的点值，用一个列表nodes记录所有不重复出现的节点。若当前节点值和dup相同，直接推进。若不同则和列表nodes的最后一个节点比较，若不同，则加入列表并推进；若相同，则把列表最后一个节点除，并把dup值设该重复节点的值，链表向前推进。最后重新连接列表中的节点即可。

- 代码：

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def deleteDuplicates(self, head):
        if not head: return head
        dup = None
        nodes = []
        c = head
        while c:
            if not dup or dup.val != c.val:
                if not nodes or nodes[-1].val != c.val:
                    nodes.append(c)
                else:
                    nodes.pop(-1)
                    dup = c
            else:
                dup = c

            c = c.next
        for i in range(len(nodes)-1):
            nodes[i].next = nodes[i+1]
        if nodes:
            nodes[-1].next = None
            return nodes[0]
        else:
            return None
```

##203. Remove Linked List Elements

- 难度：Easy
- 题目大意：

删除链表中所有值为val的节点。

- 思路：

常规删除节点的题目，需要注意若需要删除第一个节点怎么处理。这也是为什么大家通常为在链表中入头结点的原因，这样可以不用考虑第一个节点。为了让大家更清楚看到这种特殊情况，下面给出代中我没有加入头节点。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @param {integer} val
    # @return {ListNode}
    def removeElements(self, head, val):
        if not head:
            return head
        cur = head
        while cur:
            if head.val == val:
                head = head.next
                cur = head
            elif cur.val == val:
                cur = cur.next
                pre.next = cur
            else:
                pre = cur
                cur = cur.next
        return head
```

##237. Delete Node in a Linked List

- 难度：Easy
- 题目大意：

给定链表中的某个节点（非尾节点），删除该节点。

- 思路：

只给定一个节点的话，我们是没办法删除这个节点的（需要知道这个节点的前驱），但是我们可以删这个节点的next节点。观察题目，没有说不能改变节点的值，那么我们可以偷梁换柱，把该节点的值为下一个节点的值，然后删除下一个节点。

- 代码：

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def deleteNode(self, node):
        """
        :type node: ListNode
        :rtype: void Do not return anything, modify node in-place instead.
        """
        if not node or not node.next:return
        node.val = node.next.val
        node.next = node.next.next
```

#翻转链表

##206.Reverse Linked List

- 难度：Easy
- 题目大意：

翻转整个链表。

- 思路：

没什么可讲的，就地翻转必须熟记。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def reverseList(self, head):
        if head == None or head.next == None:
            return head
        pre = head
        cur = pre.next
        pre.next = None
        while cur != None:
            nex = cur.next
            cur.next = pre
            pre = cur
            cur = nex
```

```
        return pre
```

##92. Reverse Linked List II

- 难度：Medium
- 题目大意：

翻转链表中第m到n个元素。要求就地翻转，且一次遍历完成。给定的m和n总是合理的。

- 思路：

既然要求一次遍历完成，那就不能确定m,n位置然后调用上一题翻转链表函数来完成。那么我们可以第m个节点开始，翻转n-m+1个节点结束。

- 代码：

```
class Solution2:
    # @param {ListNode} head
    # @param {integer} m
    # @param {integer} n
    # @return {ListNode}
    def reverseBetween(self, head, m, n):
        if head == None or head.next == None or m == n:
            return head
        h,h.next = ListNode(-1),head
        pre,cur = h,h.next
        i = 0
        while i<=n:
            if m<=i<n:
                nex = cur.next
                cur.next = pre
                pre = cur
                cur = nex
            elif i==n:
                p1.next = pre
                p2.next = cur
            else:
                ##记录重要节点，用于拼装翻转和非翻转部分
                if i==m-1:
                    p1,p2 = pre,cur
                pre = cur
                cur = cur.next
            i+=1
        return h.next
```

##24. Swap Nodes in Pairs

- 难度：Medium
- 题目大意：

将链表中的节点两两互换。注意只能使用常数级空间，不能修改节点的值，只能使用原节点。

- 思路：

常规思路，就是做节点交换，每次推进两步，注意别越界。加入头结点，使第一个节点不用特殊处理。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def swapPairs(self, head):
        if not (head and head.next):
            return head
        h = ListNode(0)
        h.next = head
        c,n = h,head
        while n and n.next:
            c.next = n.next
            c = c.next
            n.next = c.next
            c.next = n
            c = n
            n = n.next
        return h.next
```

##25. Reverse Nodes in k-Group

- 难度：Hard
- 题目大意：

给定一个链表，以k个为一组进把链表逆序。不允许改变节点值，值运行使用常数空间。

- 思路：

这道题是Swap Nodes in Pairs的扩展版，还是常规思路就能解决，需要注意的是最后n个节点小于k，不需要逆序。为了方便操作，我们把k个节点放入列表中，然后从后往前连接next指针。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @param {integer} k
```

```
    # @return {ListNode}
    def reverseKGroup(self, head, k):
        if k == 1:
            return head
        h = ListNode(0)
        h.next = head
        c = h
        it = head
        buffer = []
        while c:
            f = 0
            buffer = []
            for i in range(k):
                if it:
                    buffer.append(it)
                    it = it.next
                else:
                    f = 1
                    break
            if len(buffer) == k:
                for i in range(k-1):
                    buffer[i+1].next = buffer[i]
            if f:
                if buffer:
                    c.next = buffer[0]
                c = None
            else:
                c.next = buffer[-1]
                c = buffer[0]
                c.next = None
        return h.next
```

#合并链表

##2.Add Two Numbers

- 难度：Medium
- 题目大意：

给定两个由链表表示的非负数字。链表中每个节点包含一个数字，数字逆序排列（地位在前，高位在），求两个数字之合，并以相同的链表格式返回。

- 思路：

这道题是一道简单的链表题，注意处理好进位，特别是最高位的进位即可。这道题本质上与合并链表想是一致的，不过合并的是值而不是节点。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
```

```
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} l1
    # @param {ListNode} l2
    # @return {ListNode}
    def addTwoNumbers(self, l1, l2):
        if not l1:
            return l2
        if not l2:
            return l1
        h1 = l1
        h2 = l2
        h3 = ListNode(-1)
        l3 = h3
        flag = 0
        while h1 or h2:
            n1 = n2 = 0
            if h1:
                n1 = h1.val
                h1 = h1.next
            if h2:
                n2 = h2.val
                h2 = h2.next
            node = ListNode((n1+n2+flag)%10)
            if n1+n2+flag >= 10:
                flag = 1
            else:
                flag = 0
            h3.next = node
            h3 = h3.next
        if flag == 1:
            node = ListNode(1)
            h3.next = node
        return l3.next
```

##21.Merge Two Sorted Lists

- 难度：Easy
- 题目大意：

将两个已排序的链表合并，注意，合并后的链表必须使用原来节点。

- 思路：

没有什么特殊的，常规思路：比较两个链表的第一个节点，取小的加入新链表，然后推进。注意头结的特殊处理，及注意别访问越界空间即可。

- 代码：

```
# Definition for singly-linked list.
```

```
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} l1
    # @param {ListNode} l2
    # @return {ListNode}
    def mergeTwoLists(self, l1, l2):
        if not l1:return l2
        if not l2:return l1
        l1c,l2c = l1,l2
        l1n,l2n = l1.next,l2.next
        if l1c.val <= l2c.val:
            head = l1c
            l1c = l1n
            if l1n:
                l1n = l1n.next
        else:
            head = l2c
            l2c = l2n
            if l2n:
                l2n = l2n.next
        c = head
        while l1c or l2c:
            if l1c and l2c:
                if l1c.val<=l2c.val:
                    c.next = l1c
                    c = c.next
                    l1c = l1n
                    if l1n:
                        l1n = l1n.next
                else:
                    c.next = l2c
                    c =c.next
                    l2c = l2n
                    if l2n:
                        l2n = l2n.next
            elif l1c:
                c.next = l1c
                break
            elif l2c:
                c.next = l2c
                break

        return head
```

##23. Merge k Sorted Lists

- 难度：Hard
- 题目大意：

将K个已排序的列表合并，注意复杂度。

● 思路：

这道题是合并2个列表的升级版，难度立马提升到Hard级别。很明显我们不能像合并2个两边那样，较k个链表的第一个节点，然后选取最小的然后推进，这样子情况太过复杂。打破最常规的思路，我能想到的思路就比较多了。

1. 我们可以把所有的节点放入数组中，然后根据节点的val进行排序，最后再把节点串起来即可。这做法实现非常简单，但是这种做法浪费了原来各个链表已经排好序这样一个条件，会使复杂度增加。

2. 创建一个大小为k的堆，把k个链表的头节点push进去。每次从堆中pop出最小的节点，若该节点有ext则加入堆中。直到堆为空。注意，若不能提供自定义排序，每次push进堆(val,node)这样的元组会方便得多。事实上，这道题考察的真是我们对堆这种数据结构的理解。

3. 分治，将问题变成合并两个已排序链表的问题。这种方法实现细节比较多，这里就不写了。

● 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution1:
    # @param {ListNode[]} lists
    # @return {ListNode}
    def mergeKLists(self, lists):
        nodes = []
        for l in lists:
            while l:
                nodes.append(l)
                l = l.next
        if not nodes:
            return
        nodes.sort(key=lambda node:node.val)
        for i in range(len(nodes)-1):
            nodes[i].next = nodes[i+1]
        nodes[-1].next = None
        return nodes[0]


class Solution2:
    # @param {ListNode[]} lists
    # @return {ListNode}
    def mergeKLists(self, lists):
        import heapq
        head=p=ListNode(-1)
        h = []
        for node in lists:
            if node: heapq.heappush(h,(node.val,node))
        while h:
            curlist = heapq.heappop(h)[1]
            p.next = curlist
```

```
        p = p.next
        if curlist.next:
            heapq.heappush(h,(curlist.next.val,curlist.next))
    return head.next
```

#改变节点顺序

此类型包括排序和其他按照题目要求改变节点相对顺序的题目。

##61. Rotate List

- 难度：Medium
- 题目大意：

给定一个链表，将右侧k个节点翻转到左侧，k为非负数。

- 思路：

这道题有两种思路，一种是用两个指针，两个指针之间保持间隔为k，当第二个指针指向尾部时，第个指针就是需要断开的地方。另一种是把节点放在列表中，这样从哪里断开就一目了然了，重新连起表就可以了。

需要注意几种特殊情况，若k大于链表长度，k=0，k=链表长度。

题目虽然为rotate，然而和翻转并没有关系。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @param {integer} k
    # @return {ListNode}
    def rotateRight(self, head, k):
        nodes = []
        while head:
            nodes.append(head)
            head = head.next
        if nodes:
            k%=len(nodes)
        left = nodes[:len(nodes)-k]
        right = nodes[len(nodes)-k:]
        if right and left:
            right[-1].next = left[0]
            left[-1].next=None
            return right[0]
        elif right and not left:
            right[-1].next=None
```

```
        return right[0]
    elif left and not right:
        left[-1].next=None
        return left[0]
    else:
        return head
```

##86. Partition List

- 难度： Medium
- 题目大意：

给定一个链表和值x，对链表进行调整，使值小于x的节点在值大于等于x的节点的左边。

- 思路：

这道题乍一看好像挺难，但其实很简单。题目表述的这个过程不就是快排的一趟调整嘛，只是变成链操作而已。当初使用链表操作会比较麻烦，要记录前驱什么的比较麻烦。但是如果把链表的节点放到表中（我很喜欢这么做！），那不就是和快排一模一样。不懂快排？没关系，说一下就懂。用i记录小x区域的右侧（第一个大于等于x的数）位置，j向前探索，若发现比x小的值，则和i位置进行交换，并把+1。

- 代码：

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

class Solution:
    # @param {ListNode} head
    # @param {integer} x
    # @return {ListNode}
    def partition(self, head, x):
        if not head:
            return head
        nodes = []
        while head:
            nodes.append(head)
            head = head.next
        i=j=0
        while j<len(nodes):
            if nodes[j].val < x:
                node = nodes.pop(j)
                nodes.insert(i,node)
                i += 1
            j += 1
        for i in range(len(nodes)-1):
            print(nodes[i].val)
            nodes[i].next = nodes[i+1]
        nodes[-1].next = None
        return nodes[0]
```

##143. Reorder List

- 难度：Medium
- 题目大意：

给定一个链表，原来的序列是L0,L1 ,...Ln-1,Ln，要求改变为L0,Ln,L1,Ln-1,L2,Ln-2...。只能使用常数空间，不可改变节点的值。

- 思路：

如果给定的不是一个链表而是一个数组就好了，因为我们需要定位到最后1个节点，且往前移动。对链表来说，往前移动是不可能的。确实可以变成数组，我最喜欢干这样的事情。但是有个问题就是，次插入都要使后面的节点往后移动，这也是数组比链表差的地方。这次我们不使用这么low的方法。们可以把问题拆分为，找链表中点，切割成两个链表，对后面那个链表原地翻转，再将两个链表合并啊哈这是一道综合题啊。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = No
class Solution:
    # @param {ListNode} head
    # @return {void} Do not return anything, modify head in-place instead.
    def reorderList(self, head):
        if not head or not head.next:return
        s1,s2=head,head
        ##cut
        while s2 and s2.next:
            s1,s2=s1.next,s2.next.next
        head2 = s1.next
        s1.next=None
        ##reverse
        head2 = self.reverseList(head2)
        ##merge
        while head2:
            p1=head.next
            head.next=head2
            head=p1
            p2=head2.next
            head2.next=head
            head2=p2


    ##206 Reverse Linked List
    def reverseList(self, head):
        if head == None or head.next == None:
            return head
        pre = head
        cur = pre.next
        pre.next = None
        while cur != None:
            nex = cur.next
```

```
            cur.next = pre
            pre = cur
            cur = nex
        return pre
```

##147. Insertion Sort List

- 难度：Medium
- 题目大意：

使用插入排序的方法对一个链表进行排序。

- 思路：

没什么好说的，人家已经把思路写在题目上了，就是要用插入排序

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param head, a ListNode
    # @return a ListNode
    def insertionSortList(self, head):
        if head == None or head.next == None:
            return head
        h = ListNode(0)
        h.next = head
        cur = head
        while cur.next:
            if cur.next.val < cur.val:
                pre = h
                while pre.next.val < cur.next.val:
                    pre = pre.next
                temp = cur.next
                cur.next = temp.next
                temp.next = pre.next
                pre.next = temp
            else:
                cur = cur.next
        return h.next
```

##148. Sort List

- 难度：Medium
- 题目大意：

使用O(n log n)时间复杂度和常数级空间复杂度对链表进行排序。

- 思路：

这回人家不告诉我们怎么做了。时间复杂度为O(nlogn)的排序算法也就那么几个，结合链表的实际情，我们采用合并排序，原因很简单，合并时空复杂度均符合，且我们之前已经写了大量的把链表切分2半，合并链表的代码，拿过来就可以直接用了。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param {ListNode} head
    # @return {ListNode}
    def sortList(self, head):
        if not head or not head.next:return head
        head2=self.cutList(head)
        return self.mergeTwoLists(self.sortList(head),self.sortList(head2))

    def cutList(self,head):
        s1,s2=head,head
        while s2.next and s2.next.next:
            s1,s2=s1.next,s2.next.next
        head2 = s1.next
        s1.next=None
        return head2

    def mergeTwoLists(self, l1, l2):
        if not l1:return l2
        if not l2:return l1
        l1c,l2c = l1,l2
        l1n,l2n = l1.next,l2.next
        if l1c.val <= l2c.val:
            head = l1c
            l1c = l1n
            if l1n:
                l1n = l1n.next
        else:
            head = l2c
            l2c = l2n
            if l2n:
                l2n = l2n.next
        c = head
        while l1c or l2c:
            if l1c and l2c:
                if l1c.val<=l2c.val:
                    c.next = l1c
                    c = c.next
                    l1c = l1n
                    if l1n:
```

```
                l1n = l1n.next
            else:
                c.next = l2c
                c =c.next
                l2c = l2n
                if l2n:
                    l2n = l2n.next
        elif l1c:
            c.next = l1c
            break
        elif l2c:
            c.next = l2c
            break
    return head
```

##328. Odd Even Linked List

- 难度：Easy
- 题目大意：

将给定列表进行重组，使奇数节点在前偶数节点在后（这里指的是序数而不是节点的值），奇数节点偶数节点）的相对位置不发生改变。要求就地解决，使用O(1)空间O(n)时间解决。

- 思路：

想法直接而简单，node.next=node.next.next，最后首尾相连。注意不要访问越界即可。

- 代码：

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution(object):
    def oddEvenList(self, head):
        """
        :type head: ListNode
        :rtype: ListNode
        """
        if not head or not head.next:return head
        h1=c1=head
        h2=c2=head.next

        while True:
            if not c1.next or not c1.next.next:
                break
            c1.next=c1.next.next
            c1=c1.next
            c2.next=c2.next.next
            c2=c2.next
        c1.next=h2
```

```
        if c2:
            c2.next=None
        return h1;
```

#环

##141. Linked List Cycle

- 难度：Medium
- 题目大意：

给定一个链表，检测其是否含有环

- 思路：

烂大街的面试题，必须会。使用两个指针，一个每次推进1步，另一个每次推进2步，若存在环，这两指针必相遇。为什么快指针不每次走3步或更多，因为快指针可能跨过慢指针而不相遇。

- 代码：

```
class Solution:
    # @param head, a ListNode
    # @return a boolean
    def hasCycle(self, head):
        s1,s2=head,head
        while s1 and s2 and s2.next:
            s1,s2 = s1.next,s2.next.next
            if s1 == s2:
                return True
        return False
```

##142. Linked List Cycle II

- 难度：Medium
- 题目大意：

给定一个链表，若存在环，求环的入口，若不存在返回null。

- 思路：

求环的升级问题，问了上面那个问题，肯定会追加这个问题的。这个问题需要进行简单的数学求证，设环之前的路程为a,相遇点离环入口x，慢指针走的路程为s，环长度为c，则存在以下关系：2s-s=nc 快指针比慢指针多走了n圈） s=a+x。两公式推导出：a=(n-1)c+c-x。则有若两个指针，一个从起点发，另一个从相遇点出发，均每次推进1步，则他们会在环入口处相遇。

- 代码：

```
class Solution:
    # @param head, a ListNode
    # @return a list node
    def detectCycle(self, head):
        s1 = self.hasCycle(head)
```

```
        if s1:
            while s1 and head:
                if s1==head:
                    return head
                s1,head=s1.next,head.next
        return None

    def hasCycle(self,head):
        s1,s2=head,head
        while s1 and s2 and s2.next:
            s1,s2 = s1.next,s2.next.next
            if s1 == s2:
                return s1
        return None
```

#综合及其他问题

##109. Convert Sorted List to Binary Search Tree （链表和树）

- 难度：Medium
- 题目大意：

给定一个升序链表，将链表变成高度平衡二叉查找树。

- 思路：

这道题其实就和链表没有什么关系了（就算有关系，我也会把节点放到列表里），主要考察的是平衡叉查找数的概念。由于要求高度平衡，因此，我们肯定总是需要查找中位数，虽然链表已经排序好了但是要查中间的节点总是需要扫一遍很不方便，因此我们把节点发到列表中（看，和链表没关系了把）。二叉查找数的定义是一个递归的过程， 因为每个节点的左子树和右子树都是二叉查找数，由于平，所以父节点总是左子树和右子树的中位数。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, x):
#         self.val = x
#         self.left = None
#         self.right = None

class Solution:
    # @param {ListNode} head
    # @return {TreeNode}
    def sortedListToBST(self, head):
```

```
        nums = []
        while head:
            nums.append(head.val)
            head = head.next
        return self.sortedArrayToBST(nums)


    def sortedArrayToBST(self, nums):
        if not nums: return None
        mid = len(nums)//2
        root = TreeNode(nums[mid])
        root.left = self.sortedArrayToBST(nums[:mid])
        root.right = self.sortedArrayToBST(nums[mid+1:])
        return root
```

##138. Copy List with Random Pointer （深度拷贝）

- 难度：Hard
- 题目大意：

给定一个链表，每个节点有一个额外的指针，指向任意链表中的节点或空。返回该链表的深度拷贝。

- 思路：

这道题的难度在于，额外的随机指针，当你复制该节点时，若随机指针指向前面的节点，如何找到，指向后面的节点，如何处理。我这里采用一种神奇的办法。复制节点的同时，把新节点插入到被复制点的后面，随机指针暂时跟被复制节点保持一致。这样一遍下来，我们构造的新链表由旧节点和对应新节点构成。再遍历一遍，我们只需要把新节点的随机指针该为其指向的节点的next即可（其指向的点为旧节点，旧节点的next是其对应的新节点）。最后遍历一遍，把新节点的next链接起来即可。

- 代码：

```
# Definition for singly-linked list with a random pointer.
# class RandomListNode(object):
#     def __init__(self, x):
#         self.label = x
#         self.next = None
#         self.random = None

class Solution(object):
    def copyRandomList(self, head):
        """
        :type head: RandomListNode
        :rtype: RandomListNode
        """
        if not head:return head
        node=head
        while node:
            copyNode = RandomListNode(node.label)
            copyNode.random = node.random
            copyNode.next = node.next
            node.next = copyNode
            node = node.next.next
```

```
        node=head
        while node:
            if node.random:
                node.next.random = node.random.next
            node = node.next.next

        phead=RandomListNode(0)
        phead.next=head
        newlist=phead
        node=head

        while node:
            phead.next=node.next
            node.next=phead.next.next
            phead=phead.next
            node=node.next
        return newlist.next
```

- 思路2：前面那种方法需要遍历3次链表，且改变了原来的数据结构，而且非常难想到那一种解决办。我们上面分析了，这道题的难点在于判断随机指针指向的节点。那么我们可以考虑用哈希表进行存节点和查询，键是旧链表节点，值是对应的新链表节点。我们第一次遍历复制每个节点，next指针和机指针暂时不管。第二次遍历以旧链表节点为键查找新的next指针和随机链表的指向。代码简洁大方哈哈哈。

```
class Solution(object):
    def copyRandomList(self, head):
        if not head:return head
        nodes={None:None}
        cur = head
        while cur:
            newCur = RandomListNode(cur.label)
            nodes[cur]=newCur
            cur = cur.next
        cur = head
        while cur:
            nodes[cur].next = nodes[cur.next]
            nodes[cur].random = nodes[cur.random]
            cur = cur.next
        return nodes[head]
```

##160. Intersection of Two Linked Lists（公共节点）

- 难度：Easy
- 题目大意：

找到两个链表公共节点的开始节点。

- 思路：

也是烂大街的面试题。先求两个链表的长度n,m，然后再用两个指针遍历，长链表指针先行abs(n-m)，后两个指针同时走，若指针指向同一节点，则该节点为公共节点的开始。

- 代码：

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, x):
#         self.val = x
#         self.next = None

class Solution:
    # @param two ListNodes
    # @return the intersected ListNode
    def getIntersectionNode(self, headA, headB):
        if not headA or not headB:return None
        la=lb=1
        ha,hb=headA,headB
        while ha.next:
            ha=ha.next
            la+=1
        while hb.next:
            hb=hb.next
            lb+=1
        if ha!=hb:return None
        if la>=lb:
            i=0
            for i in range(la-lb):
                headA=headA.next
        else:
            i=0
            for i in range(lb-la):
                headB=headB.next
        while headA!=headB:
            headA=headA.next
            headB=headB.next
        return headA
```

##234. Palindrome Linked List （回文链表）

- 难度：Easy
- 题目大意：

给定一个链表，判断是否为回文。在O(n)时间，O(1)空间内完成。

- 思路：

唉，如果是字符串或者是数组就好判断了。啊哈，这里又要祭出赖皮招式了，把节点放入列表中（这py最方便的地方，列表可以容纳一切），这样就能达到顺序和随机访问了。

- 代码：

```
# Definition for singly-linked list.
# class ListNode(object):
#     def __init__(self, x):
```

```
#         self.val = x
#         self.next = None

class Solution(object):
    def isPalindrome(self, head):
        """
        :type head: ListNode
        :rtype: bool
        """
        stack=[]
        while head:
            stack.append(head)
            head=head.next
        i,j=0,len(stack)-1
        while i<=j:
            if stack[i].val!=stack[j].val:
                return False
            i,j=i+1,j-1
        return True
```
