leetcode

20 [Valid Parentheses](https://leetcode.com/problems/valid-parentheses) （栈题目）

这道题给出一个仅包含’(‘ ‘)’ ‘[‘ ‘]’ ‘{‘ ‘}’的字符串，需要完成对该字符串内括号符号是否闭合的判断。

这道题的一道标识为easy的题，这道题的关键使用了栈的思想。分为以下个步骤：

1、初始化一个数组array

2、对字符串进行遍历操作，遇到‘(‘则向array数组push一个’)’，遇到‘[‘则向array数组push一个’]’，遇到‘{‘则向array数组push一个’}’

3、当遍历到以上三种前括号的后括号时，先检查array数组中是否为空，如果为空则闭合失败；如果不为空，则从array数组中pop一个最近存入数组的字符，如果相符，说明该后括号成功闭合，不相符，说明闭合失败。

4、当遍历结束需要检查array数组是否为空，如果为空则全部闭合成功，否则失败。代码如下。

/\*\*

\* @param {string} s

\* @return {boolean}

\*/

var isValid = function(s) {

let array = [];

for (let item of s) {

if (item === '(') {

array.push(')');

} else if(item === '[') {

array.push(']')

} else if(item === '{') {

array.push('}')

} else if(array.length === 0||array.pop()!==item) {

return false

}

}

return array.length === 0

}

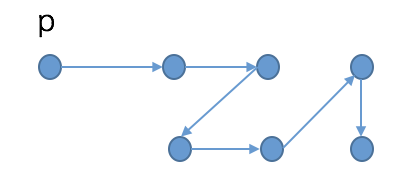
21 [Merge Two Sorted Lists](https://leetcode.com/problems/merge-two-sorted-lists)   (单链表题目)

要求：这道给出两个已排序好的单链表的头结点l1和l2，让我们合并这两个链表并将其作为一个新链表返回。新链表应该通过拼接前两个列表的节点来完成。

这道标识为easy，可以用迭代、递归两种方法来做：

先说迭代：

下列这种方法，定义了一个新的头结点并且其next会指向l1、l2中较小的那个节点。如图



1、new一个新的值为0的头结点，并且记录下该头结点，以便最后return头结点

2、遍历l1、l2，p.next 指向l1、l2中值较小的

3、当遍历到l1或l2中某一个遍历结束

4、如果l1或l2某一个还未遍历完，则将p.next直接指向l1或l2。代码如下。

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} l1

\* @param {ListNode} l2

\* @return {ListNode}

\*/

var mergeTwoLists = function(l1, l2) {

let p=new ListNode(0);

let head = p;

while(l1&&l2){

if(l1.val<l2.val){

p.next=l1;

l1=l1.next;

}else{

p.next=l2;

l2=l2.next;

}

p=p.next;

}

if(l1){

p.next=l1

}

if(l2){

p.next=l2

}

return head.next;

};

还有一种迭代的方法：

下列这种方法是将l1、l2合并到l1上，将l2上的节点插入l1中的合适位置。

1、先定义了一个new ListNode(0）节点，将该节点的next指向l1。同时定义了head节点和current节点，head节点方便最后return，定义current节点方便l2插入l1。

2、遍历l1、l2，当l1的值小于l2的值，就一直遍历l1

3、当l2的值小于l1，则将l2的值插入当前l1的前面一个

4、当遍历到l1或l2中某一个遍历结束

5、如果l1或l2某一个还未遍历完，则将cur.next直接指向l1或l2。代码如下。

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} l1

\* @param {ListNode} l2

\* @return {ListNode}

\*/

var mergeTwoLists = function(l1, l2) {

if(l1===null){

return l2;

}

if(l2===null){

return l1;

}

let head=new ListNode(0);

let cur=head;

head.next=l1;

while(l1&&l2){

if(l1.val<=l2.val){

l1=l1.next;

} else{

cur.next=l2;

let nextL2=l2.next;

l2.next=l1;

l2=nextL2;

}

cur=cur.next

}

if(l1){

cur.next=l1

}

if(l2){

cur.next=l2

}

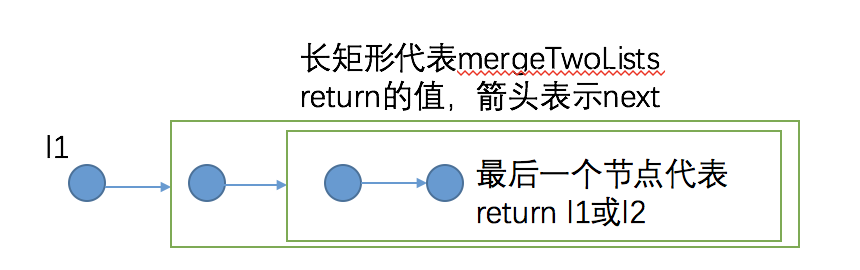
return head.next

};

最后讲一下递归方式：

这种方法也是采用了第一种迭代的思想：在第一次比较l1 l2时，哪个节点值小，则就以该节点为head，然后递归寻找下一个最小值节点，将head的next指向递归返回的节点。最终返回的就是以head为头的单链表，

这种方法取两个单链表中最小的节点，再继续递归调用mergeTwoLists合并这两个单链表的剩余节点，最后将该节点与后续节点（链表）连接起来。直到某一个链表不再有节点。



代码如下： /\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} l1

\* @param {ListNode} l2

\* @return {ListNode}

\*/

var mergeTwoLists = function(l1, l2) {

if(l1===null){

return l2;

}

if(l2===null){

return l1;

}

if(l1.val<l2.val){

l1.next=mergeTwoLists(l1.next,l2);

return l1

}else{

l2.next=mergeTwoLists(l1,l2.next);

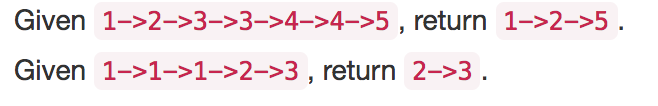
return l2;

}

};

82. Remove Duplicates from Sorted List II

要求：这道题给出一个已排序单链表的头结点，需要删除该链表中所有重复节点。如：



这道题标识为middle。

这道题目我的思路是：定义一个pre节点，pre.next指向头结点，然后移动head指针遍历单链表，如果有重复节点则，pre指向重复节点的前一个节点，head移动到最后一个重复节点的后一节点，然后将pre.next=head.next。如果没有重复节点则pre指针后移动一节点。

这道题我虽然思路正确，但是没有想清楚两点：1、怎样判断是否有重复节点；2、如果出现例如：1->2->2->3->3->4这种情况，当两个重复指针连在一起时，怎样处理pre指针的指向。

所以参照了别人的discuss。他的大体思路和我的相同，代码如下：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @return {ListNode}

\*/

var deleteDuplicates = function(head) {

let pre=new ListNode(0);

pre.next=head;

let final=pre;

while(head!==null){

while(head.next!==null&&head.val===head.next.val){

head=head.next;

}

if(pre.next===head){

pre=head;

}else{

pre.next=head.next;

}

head=head.next;

}

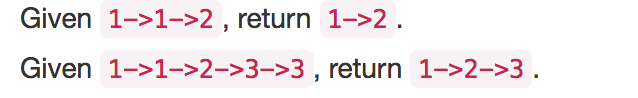
return final.next;

};

我觉得这个解决思路最妙的地方在于，使用pre.next是否等于head来判断是否存在重复节点。pre.next===head说明没有重复节点则pre向前移动一节点，并且head也向前移动一节点；如果pre.next!==head则说明存在重复节点，则将pre.next=head.next，并且head也向前移动一节点。

83. Remove Duplicates from Sorted List

要求：这道题给出一个已排序单链表的头结点，需要删除该链表中所有重复节点。



这道题标识为easy。

递归：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @return {ListNode}

\*/

var deleteDuplicates = function(head) {

if(head===null||head.next===null){

return head

}

head.next=deleteDuplicates(head.next);

return head.val===head.next.val?head.next:head;

};

迭代：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @return {ListNode}

\*/

var deleteDuplicates = function(head) {

let cur=head;

let h=head;

while(head!==null){

if(cur.val===head.val){

head=head.next;

}else{

cur.next=head;

cur=head;

head=head.next;

}

}

if(cur&&cur.next!==head){

cur.next=head;

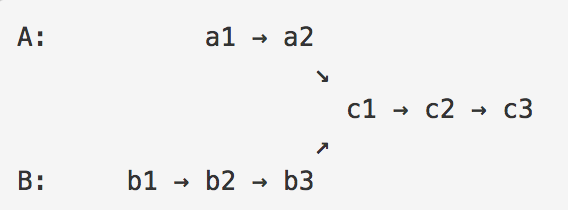
}

return h;

160. Intersection of Two Linked Lists

编程来查找两个单链表的交集的起始点。

例如，下面的两个链表开始在节点C相交。



注意：  
如果两个链表没有交集，返回null。

链表在函数返回后必须保留原来的结构。

您可以假设在整个链接结构中没有任何循环。

您的代码最好在时间复杂度为O（n），空间复杂度为O（1）。

这道题标识为easy。

这道题目我没有想出解决思路，看了这道题目前两个discuss，实现方法很优雅，但是我没有可能想到，所以我使用了第三种，我最能理解的解决方法。

1、首先计算出两个单链表的长度；

2、然后将它们对齐到相同的起始点，即让它们的长度一致；

3、最后移动这两个单链表指针，直到找到交集点或结束null为止。

代码如下：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} headA

\* @param {ListNode} headB

\* @return {ListNode}

\*/

var getIntersectionNode = function(headA, headB) {

let lengthA=calLength(headA);

let lengthB=calLength(headB);

while(lengthA>lengthB){

headA=headA.next;

lengthA--;

}

while(lengthA<lengthB){

headB=headB.next;

lengthB--;

}

while(headA!==null&&headA.val!==headB.val){

headA=headA.next;

headB=headB.next;

}

return headA;

};

function calLength(head){

let length=0;

while(head!==null){

length++;

head=head.next;

}

return length;

}

203. Remove Linked List Elements

要求：这道题给出一个单链表的头结点和一个值，如果单链表中某个节点的val等于该值则删除该节点。

这道题标识为easy。

这道题可以用递归和迭代两种方法。

递归分为层层调用removeElements方法和层层return回head.next的值。其中removeElements方法一直遍历单链表，直到head.next指向null，返回null，再层层return回去，每次return的结果是head.next的值。

递归方法代码：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @param {number} val

\* @return {ListNode}

\*/

var removeElements = function(head, val) {

if(head===null){return null}

head.next=removeElements(head.next,val);

return head.val===val?head.next:head

};

迭代方法定义了一个pre节点，用来记录每次head节点的前一个节点。

迭代方法代码：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @param {number} val

\* @return {ListNode}

\*/

var removeElements = function(head, val) {

if(head===null){return null};

let pre=new ListNode(-1);

pre.next=head;

let start=pre;

while(head!==null){

if(head.val===val){

pre.next=head.next

}else{

pre=head;

}

head=head.next;

}

return start.next;

};

206. Reverse Linked List

要求：这道题给出一个单链表的头结点，让我们使用迭代和递归两种方法返回逆置的单链表。

这道标识为easy。

这道题就是需要记录当前节点与当前节点的前一个节点。因为要将当前节点的next指向它的前一个节点。

迭代方法代码：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @return {ListNode}

\*/

var reverseList = function(head) {

let preview=null;

while(head!==null){

let next=head.next;

head.next=preview;

preview=head;

head=next

}

return preview;

};

递归方法代码：

/\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @return {ListNode}

\*/

var reverseList = function(head) {

return reverse(head,null);

};

function reverse(cur,pre){

if(cur===null){

return pre;

}

let next=cur.next;

cur.next=pre;

return reverse(next,cur);

}

这道题也说明了递归的效率比迭代高。

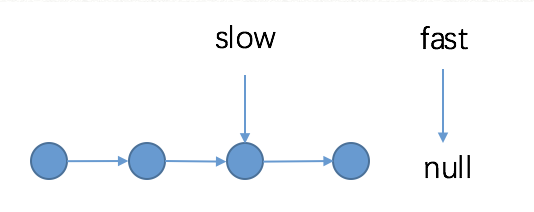
234. Palindrome Linked List

要求：这道给出一个单链表的头结点，让我们判断该单链表是否是回文。并且要求O(n) 的时间复杂度和O(1) 的空间复杂度。

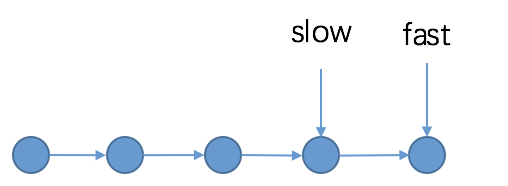
这道标识为easy。

这道题首先要找出单链表的中间位置将链表分为前部分与后部分，然后反转后部分，最后比较前部分与反转后的后部分是否一致，如果一致则是回文。详细过程如下： 1、定义两个指针slow和fast，fast指针的移动速度是slow指针的两倍，fast指针的存在是为了标识slow指针应该在何处停下，最终fast指针指向单链表末位的next，slow指针指向后部分的开始位置。

当节点为偶数个时，最终slow指针与fast指针所指位置。



当节点为奇数个时，最终slow指针与fast指针所指位置。



2、将slow开始的节点反转，返回的是反转后的头结点。

3、比较前部分与反转后的后部分是否一致，如果一致则是回文。

代码如下： /\*\*

\* Definition for singly-linked list.

\* function ListNode(val) {

\* this.val = val;

\* this.next = null;

\* }

\*/

/\*\*

\* @param {ListNode} head

\* @return {boolean}

\*/

var isPalindrome = function(head) {

let fast=head;

let slow=head;

while(fast!==null&&fast.next!==null){

fast=fast.next.next;

slow=slow.next;

}

if(fast!==null){

slow=slow.next;

}

slow=reverse(slow);

while(slow!==null){

if(slow.val!==head.val){

return false;

}

head=head.next;

slow=slow.next;

}

return true;

};

function reverse(head){

let preview=null;

while(head!==null){

next=head.next;

head.next=preview;

preview=head;

head=next;

}

return preview;

}

这道题目很值得注意的是寻找中间点的方法。