**刷题第一周总结**

**1 Reverse Linked List**

这道题应该是链表的基本操作，总结模板如下：

public class Solution {

public ListNode reverseList(ListNode head) {

//special cases

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

return head;

}

ListNode prev = null;

while(head != null){

ListNode temp = head.next;

head.next = prev;

prev = head;

head = temp;

}

return prev;

}

}

**注意：**

1 本题的基本思路是：通过两个指针，一个指针pre指向将要改变节点的前一个节点，另一个指针指向将要改变的节点。通过这两个指针遍历整个链表，将链表所有节点连接的“方向”反过来。

2 链表的末尾一般是null，所以pre指针初始时指向null。

3 遍历结束时，处于队头的是pre指针，作为返回值返回。

4 程序最开始将链表为空和只有一个节点的特殊情况单独列出，直接返回，但实际上，循环体本身也可以正确处理这两种情况。

5 此题应该当作链表题目的基本模板。

6 时间复杂度O(n), 空间O(1)

此题还有一种递归解决的方法：

public class Solution {

public ListNode reverseList(ListNode head) {

//base case

if(head == null){

return head;

}

if(head.next == null){

return head;

}

ListNode post = head.next;

head.next = null;

ListNode reverseRest = reverseList(post);

post.next = head;

return reverseRest;

}

}

**2 Remove Nth Node from End of List**

此题的主要问题在于只用一次遍历如何获得从表尾数第n个节点的前一个节点，采用典型的“快慢指针法”:

public class Solution {

public ListNode removeNthFromEnd(ListNode head, int n) {

//special cases

if(head == null || n <= 0){

return head;

}

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode slow = dummy;

ListNode fast = dummy;

for(int i = 0; i < n; i++){

if(fast == null){ 1

return head; 2

}

fast = fast.next;

}

while(fast.next != null){

fast = fast.next;

slow = slow.next;

}

ListNode post = slow.next.next; 3

slow.next = post; 4

return dummy.next;

}

}

**注意：**

1 采用**”**快慢指针法“定位到所要寻找节点的前一个节点，此方法在最后有总结。有了要找节点的上一个节点，就可以改变next指针从而”删除“指定的节点。

2 由于链表的头节点head可能改变，要在表头加入dummy节点，此技巧在最后有总结。

3 此题中说n值一定是合法的，但是也应该想想如果它可能不合法该怎么办。1，2两句应对n比链表长度还大的情况，此时直接返回。

4 想要删除节点，在java中，实际上就是让该节点的前一个节点的next指针，指向删除节点的下一个节点（pre.next.next）， 即3，4句。

5 时间O(n),空间O（1）

**3 Remove linked list element:**

这道题是典型的链表操作，采用一个指针遍历整个list，直到list的末尾，每次检查的是指针的下一个节点的值:

public class Solution {

public ListNode removeElements(ListNode head, int val) {

//special cases

if(head == null){

return head;

}

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode prev = dummy;

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

if(prev.next.val == val){

ListNode temp = prev.next.next;

prev.next = temp;

} else {

prev = prev.next;

}

}

return dummy.next;

}

}

**注意：**

1 头节点head可能改变，所以要使用dummy指针。

2 多个值等于val的节点可能连续存在，所以此题中pre指针只有在确定下一个节点的值不等于val时才会向后移，相当于”pre永远指向已检查过的值不是val的最后一个节点”。

3 时间O(n),空间O(1)

**4 Remove duplicates from sorted list**

典型的链表操作题目，由于链表已排序，且只删掉重复的元素，因此使用一个指针遍历list。当发现当前节点的值与下一个相同时，删除下一个节点：

public class Solution {

public ListNode deleteDuplicates(ListNode head) {

//special cases

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

return head;

}

ListNode cur = head;

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

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

ListNode temp = cur.next.next;

cur.next = temp;

} else {

cur = cur.next;

}

}

return head;

}

}

**注意：**

1 while循环的条件中不能忘记cur.next != null这句，因为循环体中要检查cur.next.val，当cur指向最后一个节点时，这句话会报错因为cur.next没有val这个变量。

2 时间O(n), 空间O(1)

**5 Merge two sorted lists**

这也应该算是链表题目的基本操作模板：

public class Solution {

public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

ListNode dummy = new ListNode(0);

ListNode tail = dummy;

while(l1 != null && l2 != null) {

if(l1.val < l2.val){

tail.next = l1;

l1 = l1.next;

} else {

tail.next = l2;

l2 = l2.next;

}

tail = tail.next;

}

if(l1 != null){

tail.next = l1;

}

if(l2 != null){

tail.next = l2;

}

return dummy.next;

}

}

**注意：**

1 tail指针永远指向已经merge好的list的最后的一个节点，每一步基本操作是判断两个待merge的list的当前节点值大小，将tail.next指向小的那一个，再将对应的指针向后挪一位。

2 头节点无法确定是谁，所以使用dummy指针。

3 循环体的条件是两个待归并的list的指针都不为空（即都未到末尾）。

4 当一个list到达末尾，另一个可能还没排完，所以最后要检查两个list的指针是否不为空，不为空的话，把这个list剩下的元素补在新list结尾。

5 时间O(m + n),空间O(1)

**6 Intersection of two linked list:**

这道题有两种办法：

1. 双指针法：

先通过两次遍历获得两个list的长度，之后通过两个长度做差，获得两个list的长度差，将较长list的指针先后挪这个长度，之后两个指针一起后挪，当发现两个指针指向同一个节点，返回即可。

1. 更巧妙的方法：

实际上是用一种巧妙的方法实现上述过程。我们假设list1比list2长，当list2的指针指到null的时候，我们把它重新付给list1的head，当list1的指针抵达null结尾时，我们把它给list2的head，而此时，list1的指针一定比list2的指针先走了（lengthA - LengthB）步:

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

//special cases

if(headA == null || headB == null){

return null;

}

ListNode h1 = headA;

ListNode h2 = headB;

while(h1 != h2){

h1 = h1 == null? headB:h1.next;

h2 = h2 == null?headA:h2.next;

}

return h1;

}

}

注意：

1 如果两个list有一个为空，则可以直接返回。

2 时间O(m + n)，空间O(1)

**7 add two numbers**

方法比较容易想到，就是从表头起诸位相加进位。但是细节很多，尤其注意两个list很可能不等长。我的代码如下：

public class Solution {

public ListNode addTwoNumbers(ListNode l1, ListNode l2) {

//special cases

if(l1 == null && l2 == null){

return null;

}

int up = 0;

ListNode head = null;

ListNode walker = null;

while(l1 != null || l2 != null){

ListNode node = new ListNode(0);

int add1 = l1 == null?0:l1.val; 1

int add2 = l2 == null?0:l2.val; 2

node.val = (add1 + add2 + up)%10;

if(head == null){

head = node;

} else {

walker.next = node;

}

walker = node;

up = (add1 + add2 + up)/10;

l1 = l1 == null?l1:l1.next; 3

l2 = l2 == null?l2:l2.next; 4

}

if(up != 0){

ListNode node = new ListNode(up);

walker.next = node;

walker = node;

}

//walker.next = null;

return head;

}

}

注意：

1. 1，2，3，4四句就是为处理一个list比另一个提前结束而写。
2. 在循环结束后要再检查进位up是否为0，不为0的话需要再加一个结点。
3. 时间O(n),空间O(1)

**8 Swap nodes in pairs:**

典型的链表题目，依然通过一前一后两个指针遍历整个链表实现:

public class Solution {

public ListNode swapPairs(ListNode head) {

//special case

if(head == null){

return head;

}

ListNode dummy = new ListNode(0);

dummy.next = head;

ListNode pre = dummy;

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

ListNode temp = head.next.next;

ListNode post = head.next;

pre.next = post;

post.next = head;

head.next = temp;

pre = head;

head = temp;

}

return dummy.next;

}

}

注意：

1. 由于头节点会更改，所以加入dummy指针以便于处理边界情况。
2. 由于循环体中要获得当前节点之后的第二个节点，所以许还条件要包含head.next != null，当到达倒数第二个结点并完成调换后，即结束
3. 时间O(n),空间O(1)

**9 Rotate list:**

本题同样要采用”快慢指针法“，从而得到移位之后表尾的元素位置。之后通过将原list首尾相连再断开，获得新的rotate之后的list:

public class Solution {

public ListNode rotateRight(ListNode head, int k) {

//special case

if(head == null || k == 0){

return head;

}

//get the length of the list

ListNode walker = head;

int count = 1;

while(walker.next != null){

walker = walker.next;

count++;

}

k = k % count;

ListNode dummy = new ListNode(0);

dummy.next = head;

//find the breaking point

walker = head;

ListNode runner = head;

for(int i = 0; i < k; i++){

runner = runner.next;

}

while(runner.next != null){

walker = walker.next;

runner = runner.next;

}

//make it a ring

runner.next = dummy.next;

dummy.next = walker.next;

walker.next = null;

return dummy.next;

}

}

注意：

1. k的值可能大于list的长度，所以，要首先获得list长度，之后让k对长度取模，从而保证k值是小于list长度的。
2. 本题的首结点会改变，因此加入dummy指针.
3. 快慢指针定位到新的尾结点之后，首先将list首尾相接形成环，之后将dummy指针的next指向尾结点之后的结点，最后把新的尾结点的next赋成null。这样，就完成了”rotate”.
4. 时间O(n),空间O(1)

**10 sort list**

本题综合应用到了链表的快慢指针法，归并操作，递归算法和merge sort算法。首先，此题要求时间复杂度为nlogn级别，这种复杂度的常见排序算法就是快速排序和归并排序。而快速排序需要频繁改变list，比较麻烦，最容易实现的是merge sort。对于merge sort，最重要的在于两部分，一是归并过程，这个就与第五题基本类似，还有就是如何寻找一个list的中点，这需要采用“快慢指针法“:

public class Solution {

public ListNode sortList(ListNode head) {

//base case

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

return head;

}

ListNode mid = findMiddle(head);

ListNode right = sortList(mid.next);

mid.next = null;

ListNode left = sortList(head);

return merge(left, right);

}

//merge two lists

public ListNode merge(ListNode head1, ListNode head2){

ListNode dummy = new ListNode(0);

ListNode tail = dummy;

while(head1 != null && head2 != null){

if(head1.val < head2.val){

tail.next = head1;

tail = head1;

head1 = head1.next;

} else {

tail.next = head2;

tail = head2;

head2 = head2.next;

}

}

if(head1 != null){

tail.next = head1;

}

if(head2 != null){

tail.next = head2;

}

return dummy.next;

}

//find the middle node of the list

public ListNode findMiddle(ListNode head){

ListNode slow = head;

ListNode fast = head.next;

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

slow = slow.next;

fast = fast.next.next;

}

return slow;

}

}

注意：

1 merge 函数需要返回merge之后序列的头指针，每次递归调用sortList()都要返回。

2 时间O(nlogn),空间由于用到递归系统栈，所以不是O(1),不知有谁知道是多大级别的？

**链表题目综合总结：**

链表的特点就是可以任意延长加入新元素，但是只能从表头逐个遍历整个list以访问元素。我认为，以上十道题中2，5，6，9，10题可以算作一类，他们都使用了一些简便的技巧处理链表。剩下的1，3，4，7，8题则均采用一个或多个指针遍历链表修改连接结构。在链表题目中，有几个常见的处理技巧，总结如下：

**1 快慢指针法：**

两个指针，或一前一后，或一快一慢遍历list，当处于前方的指针抵达表尾时，较慢的指针往往处在某个特殊的位置，比如获得距表尾第n位结点的前一个结点：

ListNode slow = head;

ListNode fast = head;

for(int i = 0; i < n; i++){

if(fast == null){ 1

return head; 2

}

fast = fast.next;

}

while(fast.next != null){

fast = fast.next;

slow = slow.next;

}

让快指针先走n步。再比如，获得list的中点结点，让快指针比慢指针速度快一倍：

ListNode slow = head;

ListNode fast = head.next;

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

slow = slow.next;

fast = fast.next.next;

}

return slow;

**2反转链表：**

ListNode prev = null;

while(head != null){

ListNode temp = head.next;

head.next = prev;

prev = head;

head = temp;

}

return prev;

1. **dummy指针：**

有时，链表的头节点head指向的元素可能会改变，由于是第一个节点，没有前结点，代码的处理方法和其他节点会不一样，这就使其变为了边界情况，为了保持代码对于每一个节点的通用性，我们在head之前加入一个dummy结点，值设为0，dummy.next指向head，让遍历list的指针从它开始后移，这样，头节点也变成和其他节点一样了。注意再函数需要返回头结点时，返回dummy.next。