合并两个或K个排序的列表\_LeetCode\_21\_Easy\_23\_Hard\_剑指Offer\_25

# 合并两个排序的列表

## 题目描述

/\*\*\*\*\*\*\*\*\*\*\*\*题目1：合并两个有序的链表<Leetcode\_21与JianzhiOffer\_25>\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\*\*

\* 难度：Easy

\* DateTime：2018-10-05

\* https://leetcode.com/problems/merge-two-sorted-lists/description/

\* 题目描述：

\* 输入两个单调递增的链表，输出两个链表合成后的链表，当然我们需要合成后的链表满足单调不减规则。

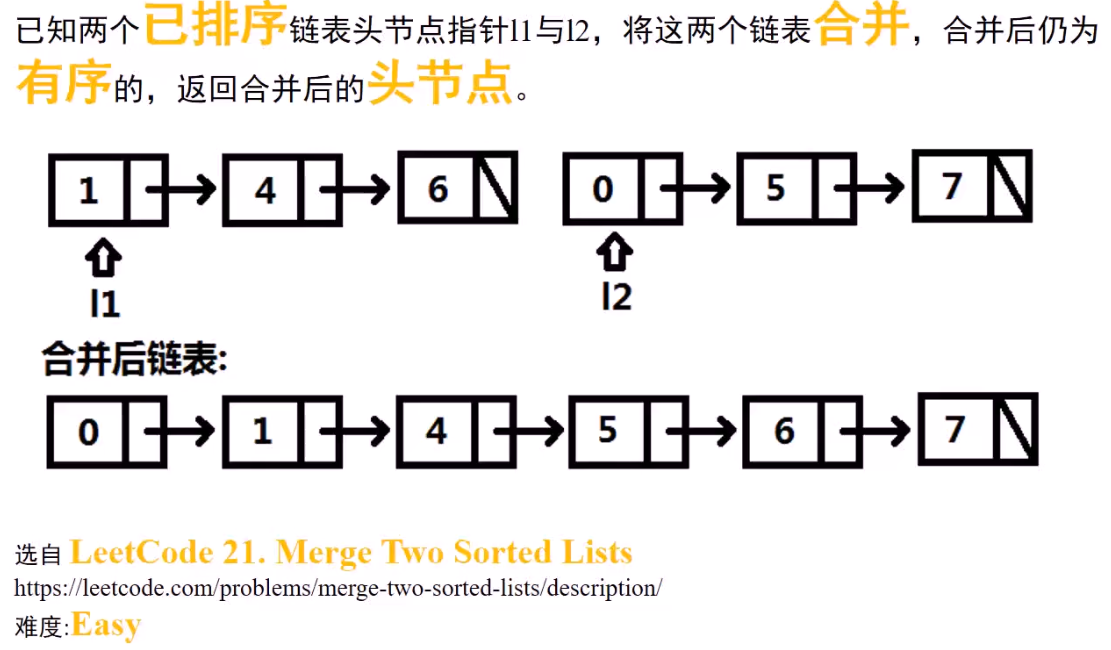
\* Merge two sorted linked lists and return it as a new list.

\* The new list should be made by splicing together the nodes of the first two lists.

\* Example:

\* Input: 1->2->4, 1->3->4

\* Output: 1->1->2->3->4->4



## 思路分析

\* 思路分析：

\* **方法1：递归思想**。不断地获取合并链表的下一节点，直到原两个链表访问到null。

\* 递归终止条件：链表其中之一为null;

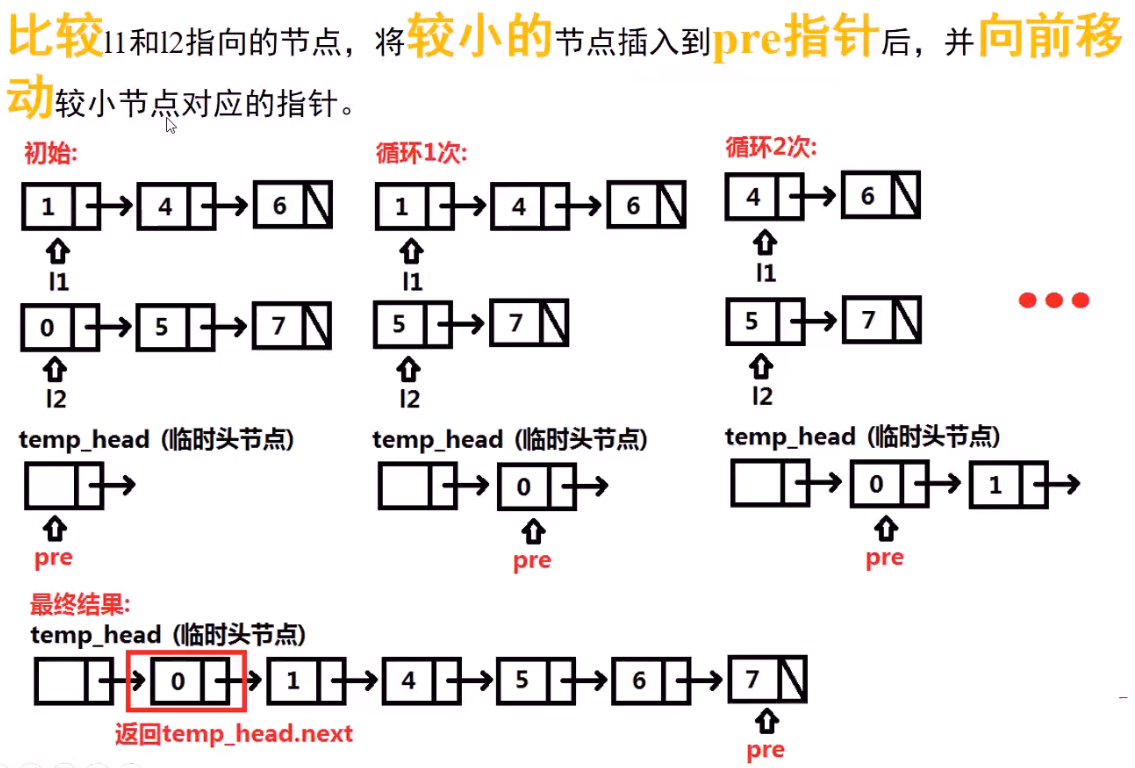
\* **方法2：循环思想**。

\* 当两个链表都非null时，while循环不断判断；

\* 当其中一个链表为null，通过if语句判断出那个为null，然后将非null链表接到合并链表上即可。

\* 复杂度分析：假设两个链表的长度分别为N和M，则在最坏的情况下，需要比较N+M-1次,即复杂度可以认为是O（N+M）。

\* 如1->3; 2->4; 第一次比较1和2，得到1；第二次比较2和3，得到2；第三次比较3和4，得到3；最后得到4.



## Java代码

### 方法1：递归方法

/\*\*

\* 方法1：递归方法

\* 2018-08

\*/

public ListNode mergeTwoLists\_1(ListNode list1, ListNode list2) {

if (list1 == null) return list2;

if (list2 == null) return list1;

ListNode mergeHead = null;

if (list1.val <= list2.val) {

mergeHead = list1;

mergeHead.next = mergeTwoLists\_1(list1.next, list2);

} else {

mergeHead = list2;

mergeHead.next = mergeTwoLists\_1(list1, list2.next);

}

return mergeHead;

}

/\*\*

\* 方法1：递归方法<更简洁的代码>

\* 2018-10

\*/

public ListNode mergeTwoLists\_2(ListNode l1, ListNode l2) {

if (l1 == null) return l2;

if (l2 == null) return l1;

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

l1.next = mergeTwoLists\_2(l1.next, l2);

return l1;

} else {

l2.next = mergeTwoLists\_2(l1, l2.next);

return l2;

}

}

### 方法2：while循环方法

/\*\*

\* 方法2：非递归方法while实现

\* 2018-10-05 北邮教三

\*/

public ListNode mergeTwoLists(ListNode list1, ListNode list2) {

ListNode mergedListHead = new ListNode(-1);//初始化一个头节点

ListNode head1 = list1, head2 = list2;

ListNode temp = mergedListHead;//行走的node

//两个链表都非null时

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

if (head1.val <= head2.val) {

temp.next = head1;

head1 = head1.next;

} else {

temp.next = head2;

head2 = head2.next;

}

temp = temp.next;

}

//一个或两个都为null时

if (null == head1)

temp.next = head2;

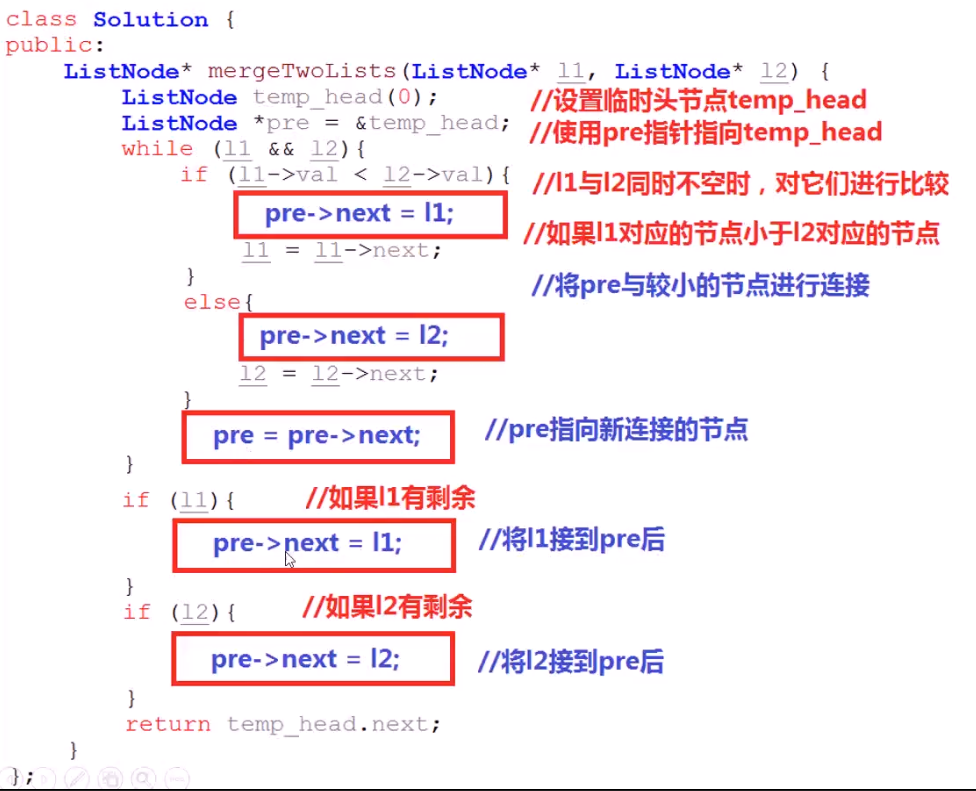
if (null == head2)

temp.next = head1;

return mergedListHead.next;

}

### C语言代码



# 合并k个链表\_Hard

## 题目介绍

\* 难度:**Hard**

\* DateTime:2018-10-05 10:29 北邮教三

\* 链接：

\* 题目描述：https://leetcode.com/problems/merge-k-sorted-lists/description/

\* Merge k sorted linked lists and return it as one sorted list.

\* Analyze and describe its complexity.<重点：会分析复杂度>

\* Example:

\* Input:

\* [

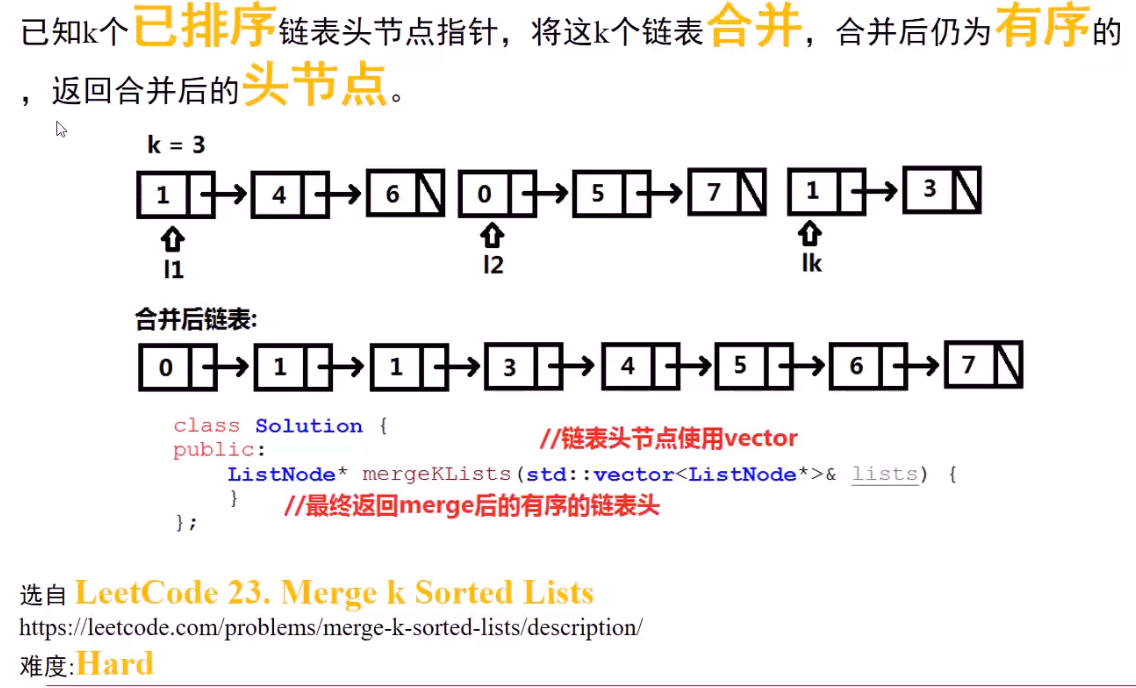
\* 1->4->5,

\* 1->3->4,

\* 2->6

\* ]

\* Output: 1->1->2->3->4->4->5->6



## 思路分析

\* 复杂度分析的前提：长度为N和M的两个链表，合并在一起最坏情况下时间复杂度为O（N+M）。

\*

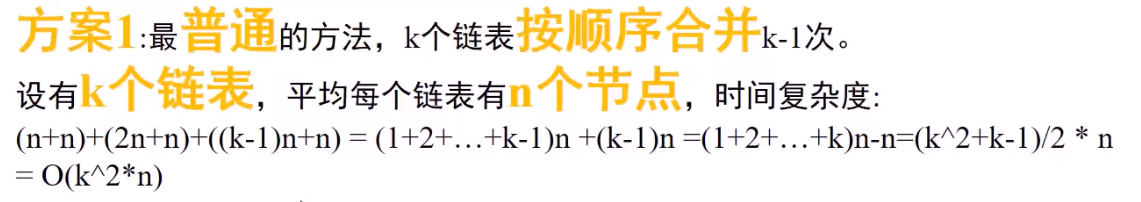
\* 方法1：先合并2个，然后将结果再与一个新链表合并，依次类推，直至合并完毕。

\* 假设4个链表A、B、C、D，首先合并A、B得到AB，

\* 然后合并AB、C，得到ABC，最后合并ABC、D，得到ABCD。

\* 复杂度分析：假设每个链表长度为N，共k个链表，则

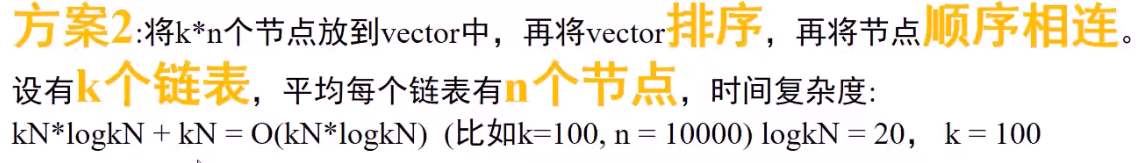
\* 2N+3N+4N+...+kN=((k\*k+k-2)/2)\*N=O(k\*k/2\*N);



\* 方法2：将这些链表的节点存放到list中，然后调用list的sort方法，指定根据val大小排序的比较器，

\* 进行排序，最后将这些节点依次连在一起即可。

\* 时间复杂度：就是排序的时间复杂度(N\*logN):(k\*N)\*log(k\*N);



\* 方法3：利用分治策略。

\* 合并k个链表，只需要先分成2组，先合并k/2个链表，

\* 同样，合并k/2个链表，也分成2组，先合并k/4个链表；...

\* 最终，只需要2个链表即可，然后再依次返回。

\* 利用递归的方法解决。

\* 时间复杂度分析：k个链表，每个链表长度为N，一共可以分治成logk层。

\* 第一层，只需要合并1次，每合并1次，时间复杂度为O(k\*N/2+k\*N/2)=O(k\*N);

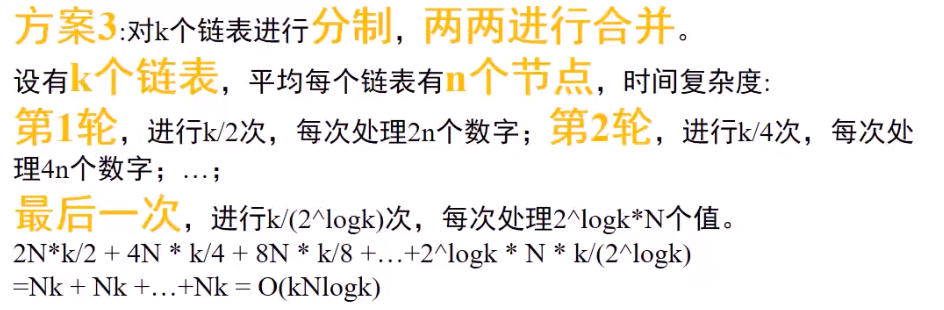
\* 第二层，需要合并2次，每合并1次，时间复杂度为O（k\*N/4+k\*N/4）,则该层复杂度为2\*O（k\*N/2）=O(k\*N);

\* 第三层，需要合并4次，每合并1次，时间复杂度为O（k\*N/8+k\*N/8）,则该层复杂度为4\*O（k\*N/4）=O(k\*N);

\* ......

\* 第logk层，需要合并2^(logk -1)=k/2次，每合并1次，时间复杂度为O（2N），则该层复杂度为k/2\*O(2N)=O(k\*N).

\* 因此可以得出，每一层的时间复杂度都是O(k\*N);logk层，因此最终时间复杂度为O(k\*N\*logk).



\*

\* 三种算法的时间复杂度相比较：O(k\*k\*N)>(k\*N)\*log(k\*N)>O(k\*N\*logk);

\* 因此，第三种分治算法最优。

\* 注意：方法1和方法3都需要利用mergeTwoLists方法。

## Java代码

### 算法1：逐个合并

/\*\*

\* 算法1：逐个合并。

\* Leetcode Submission Result: Accepted

\*/

public ListNode mergeKLists\_1(ListNode[] lists) {

if(null == lists||lists.length == 0) return null;

ListNode mergedList = lists[0];

for(int i = 1;i < lists.length;i++){

mergedList = mergeTwoLists(mergedList,lists[i]);

}

return mergedList;

}

### 对节点排序

/\*\*

\* 算法2：对节点排序。

\* Submission Result: Accepted

\*/

public ListNode mergeKLists\_2(ListNode[] lists) {

if(null == lists||lists.length == 0) return null;

ArrayList<ListNode> nodeList = new ArrayList<ListNode>();

for(ListNode head:lists){//将所有链表的节点都添加到nodeList中

while(null != head){

nodeList.add(head);

head = head.next;

}

}

if(nodeList.size() == 0) return null;//排除所有链表都是null的特殊情况,否则不通过

nodeList.sort(new Comparator<ListNode>() {

@Override

public int compare(ListNode o1, ListNode o2) {

return o1.val-o2.val;

}

});

for(int i = 1;i < nodeList.size();i++){

nodeList.get(i-1).next = nodeList.get(i);

}

return nodeList.get(0);

}



### 分治策略

/\*\*

\* 算法3：分治策略与递归实现。divideAndConque

\* Submission Result: Accepted

\* 代码分成三部分：

\* 1. 递归终止条件；

\* 2. 分割ListNode数组为两个子数组；

\* 3. 递归调用分别合并子数组，然后再利用mergeToList合并递归方法返回的链表。

\*/

public ListNode mergeKLists\_divideAndConque(ListNode[] lists) {

//第一部分

if(null == lists||lists.length == 0) return null;

if(lists.length == 1) return lists[0];

//第二部分

int num = lists.length;

int mid = num/2;

ListNode[] subLists1 = new ListNode[mid];

ListNode[] subLists2 = new ListNode[num-mid];

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

subLists1[i] = lists[i];

}

for(int i = mid;i < num;i++){

subLists2[i-mid] = lists[i];

}

//第三部分

ListNode mergedList1 = mergeKLists\_divideAndConque(subLists1);

ListNode mergedList2 = mergeKLists\_divideAndConque(subLists2);

return mergeTwoLists(mergedList1,mergedList2);

}



# 题目3、 LeetCode\_148\_SortList\_Medium(利用题目1的结果+双指针均分链表)

## 题目介绍

\* 题目3、 LeetCode\_148\_SortList\_Medium(利用题目1的结果+双指针均分链表)

\* https://leetcode.com/problems/sort-list/description/

\* 难度：Medium

\* DateTime:2018-10-14 北邮教三

\* <p>

\* 题目介绍：

\* Sort a linked list in O(n log n) time using constant space complexity.

\* Example 1:

\* Input: 4->2->1->3

\* Output: 1->2->3->4

\* <p>

\* Example 2:

\* Input: -1->5->3->4->0

\* Output: -1->0->3->4->5

\* <p>

## 思路分析

\* 思路分析：

\* 要使时间复杂度为O(NlogN);显然需要将不断分割一半一半的策略，因此想到了归并和快排。

\* 1. 对于归并方法：merge方法。

\* 首先将链表递归地逐渐分成一半一半，然后分别对一半一半排序，逐渐变成两个有序链表，

\* 再利用LeetCode\_21合并有序链表的方法将两个有序链表合并为一个有序链表。

\* 难点：如何将一个链表变成两个链表？

\* 方法:利用双指针的方法。

\* 前提：如何合并两个有序链表？熟练掌握。

\* 2. 基于快排方法：partition方法。

\* 指定第一个节点为pivot，然后遍历后序节点，将val小于pivot的节点插入到pivot之前，

\* val大于pivot的节点插入到pivot之后;最终返回pivot节点。

\*

\* 总结：经过大量测试，归并方法效率明显高于快排，

\* LeetCode提交归并耗时5ms;而快排耗时500多毫秒。在Discuss讨论区，也是以归并方法为主。

## Java代码

/\*\*

\* 1. 基于归并方法

\*/

public ListNode sortList(ListNode head) {

if (head == null || head.next == null) return head;

//利用双指针平均分割链表

ListNode slow = head;

ListNode fast = head.next;//fast在前一步，目的是让slow最后指向的节点是前一段的最后一个

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

slow = slow.next;

fast = fast.next.next;

}

ListNode subH2 = slow.next;//第二个子链表以slow.next为头节点

slow.next = null;//断开:容易忽略

//对分割后的子链表进行排序

ListNode sortH1 = sortList(head);//对前一个子链表排序

ListNode sortH2 = sortList(subH2);//对后一个子链表排序

//合并两个有序子链表

return mergeTwoLists\_Recursive(sortH1, sortH2);//调用上面的方法

}

/\*\*

\* 2. 基于快方法

\*/

public ListNode sortList\_2(ListNode head) {

if (head == null || head.next == null) return head;

ListNode dummy = new ListNode(-1);

dummy.next = head;

quickSort(dummy, null);

return dummy.next;

}

/\*\*

\* 基于快排算法的方法

\*

\* @param dummyHead:傀儡头节点

\* @param dummyTail:傀儡尾节点

\*/

public void quickSort(ListNode dummyHead, ListNode dummyTail) {

ListNode midNode = partition(dummyHead, dummyTail);

if (dummyHead.next != midNode) quickSort(dummyHead, midNode);//递归前一段

if (midNode.next != dummyTail) quickSort(midNode, dummyTail);//递归后一段

}

/\*\*

\* 将以dummyHead开头以dummyTail结尾的链表以pivot节点为中心，按照val大小分成两部分，

\* 前边的小于pivot的值，右边的节点值大于等于pivot的值。

\*

\* @param dummyHead : 傀儡头节点

\* @param dummyTail : 傀儡尾节点

\*/

public ListNode partition(ListNode dummyHead, ListNode dummyTail) {

ListNode pivot = dummyHead.next;//基准节点

ListNode dummyPre = dummyHead.next;//遍历节点的前一节点,用于删除当前节点，利用headPre.next表示currNode

while (dummyPre.next != dummyTail) {//不对dummyTail排序,dummyTail也是一个傀儡

if (dummyPre.next.val < pivot.val) {

//首先移出preNode.next节点

ListNode moveNode = dummyPre.next;

dummyPre.next = moveNode.next;//移出了preNode.next

//然后插入到headHead之后

moveNode.next = dummyHead.next;

dummyHead.next = moveNode;

} else {

//如果dummyPre.next节点的值大于等于pivot的值，此时不需要移动节点，只需要移动dummyPre即可

dummyPre = dummyPre.next;

}

}

return pivot;

}