Branch: master **OOP-Design** / ******LinkedListSummary.java Find file Copy path

huyilong SUMMARY - ALLLL **LinkedList** Problems 0d8a413 32 seconds ago

1 contributor

1179 lines (1018 sloc) | 38.7 KB Raw Blame History

```java
 class Node{
    Node pre;
    Node next;
    int key;
    int value;

    public Node(int key, int value){
        this.key = key;
        this.value = value;
    }
}
public class LRUCache {
    private int capacity;
    //both has a hashmap and a doubly-linked list
    private HashMap<Integer, Node> map = new HashMap<>();
    private Node head = new Node(-1,-1);
    private Node tail = new Node(-1,-1);

    public LRUCache(int capacity){
        this.capacity = capacity;
        head.next = tail;
        tail.pre = head;
        //always just from back to begin
        //and then from begin to back
    }

    public int get(int key){
        //here we are getting the values
        if(!map.containsKey(key)){
            return -1;
        }else{
            //we found the key and need to return the value
            //&& put this node at the tail of the linkedlist
            Node used = map.get(key);
            //remove the node!!!!!
            //we have the code to move it to tail later
            //just break it here
            used.pre.next = used.next;
            used.next.pre = used.pre;

            moveTail(used);

            return used.value;

        }
    }
    public void set(int key, int value){
        //we firstly need to check whether get has the valid return
        if(get(key) != -1){
            //no need to create a new node
            //there already exists one for update
            map.get(key).value = value;
            return;
            //here just done
        }

        if(map.size() == capacity){
            //also we need to remove the node in the hashmap
            map.remove(head.next.key);
            //remove hashmap by key is just this easy
            //we could directly remove(key)

```

```
            //we need to remove the node in the beginning at first
            //before we insert the new node into the tail of the list -- avoid overflow
            //
            //remove the first node after dummy head
            //and insert the node before the dummy tail
            head.next.next.pre = head;
            head.next = head.next.next;
        }
        Node newNode = new Node(key, value);
        map.put(key, newNode);
        moveTail(newNode);
    }

    private void moveTail(Node used){
        //from back to front
        used.pre = tail.pre;
        tail.pre.next = used;

        tail.pre = used;
        used.next = tail;
        //from front to back

    }
}
```

这道题考察了基本的链表操作，注意当改变指针连接时，要用一个临时指针指向原来的next值，否则链表丢链，无法找到下一个值。
需要运用fakehead来指向原指针头，防止丢链，用两个指针，ptr1始终指向需要交换的pair的前面一个node，
ptr2始终指向需要交换的pair的第一个node。

```
ListNode cur = head; ListNode pre= null;
        while(cur!=null){
            ListNode next = cur.next;
            cur.next = pre;

            pre = cur;
            cur = next;
        }
return pre;
```

需要用一个临时指针nextstart， 指向下一个需要交换的pair的第一个node，保证下一次交换的正确进行。

然后就进行正常的链表交换，和指针挪动就好。

当链表长度为奇数时，ptr2.next可能���null；

当链表长度为偶数时，ptr2可能为null。

所以把这两个情况作为终止条件，在while判断就好，最后返回fakehead.next

```
public ListNode swapPairs(ListNode head) {
      if(head == null || head.next == null)
        return head;

      ListNode fakehead = new ListNode(-1);
      fakehead.next = head;

      ListNode ptr1 = fakehead;
      ListNode ptr2 = head;

      while(ptr2!=null && ptr2.next!=null){
        需要用一个临时指针nextstart， 指向下一个需要交换的pair的第一个node
          ListNode nextstart = ptr2.next.next;
          ptr2.next.next = ptr2;
          ptr1.next = ptr2.next;
          ptr2.next = nextstart;
          ptr1 = ptr2;
          ptr2 = ptr2.next;
      }
    return fakehead.next;
  }
```

Given an unsorted array nums, reorder it in-place such that nums[0] <= nums[1] >= nums[2] <= nums[3]....

For example, given nums = [3, 5, 2, 1, 6, 4], one possible answer is [1, 6, 2, 5, 3, 4].

排序法 复杂度 时间 O(NlogN) 空间 O(1)

思路 根据题目的定义，摇摆排序的方法将会很多种。我们可以先将数组排序，
这时候从第3个元素开始，将第3个元素和第2个元素交换。然后再从第5个元素开始，
将第5个元素和第4个元素交换，以此类推。就能满足题目要求。

```
public class Solution {
```

```
    public void wiggleSort(int[] nums) {
        // 先将数组排序
        Arrays.sort(nums);
        // 将数组中一对一对交换
        for(int i = 2; i < nums.length; i+=2){
            int tmp = nums[i-1];
            nums[i-1] = nums[i];
            nums[i] = tmp;
        }
    }
}

交换法 复杂度 时间 O(N) 空间 O(1)

思路

题目对摇摆排序的定义有两部分：

如果i是奇数，nums[i] >= nums[i - 1]
如果i是偶数，nums[i] <= nums[i - 1]
所以我们只要遍历一遍数组，把不符合的情况交换一下就行了。
具体来说，如果nums[i] > nums[i - 1]， 则交换以后肯定有nums[i] <= nums[i - 1]。

public class Solution {
    public void wiggleSort(int[] nums) {
        for(int i = 1; i < nums.length; i++){
            // 需要交换的情况：奇数时nums[i] < nums[i - 1]或偶数时nums[i] > nums[i - 1]
            if((i % 2 == 1 && nums[i] < nums[i-1]) || (i % 2 == 0 && nums[i] > nums[i-1])){
                int tmp = nums[i-1];
                nums[i-1] = nums[i];
                nums[i] = tmp;
            }
        }
    }
}
---------------1.fast/slow to find mid/kth 2.reverse linkedlist as needed-------
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ListNode reverseList(ListNode head) {
        only one node -- no need to reverse
        if(head == null || head.next == null){
            return head;
        }

        ListNode cur = head;
        ListNode pre= null;
        while(cur!=null){
            ListNode next = cur.next;
            cur.next = pre;

            pre = cur;
            cur = next;
        }
        return pre;
    }
    public ListNode reverseList(ListNode head) {
        //recursion
        //check input just
        if(head == null){
            return null;
        }

        //this is truly useful base condition
        if( head.next == null){
            //System.out.println(head.val);
            return head;
        }

        //reverseList(head.next);
        //we should do
        ListNode newHead = reverseList(head.next);
        //System.out.println(head.val);
        head.next.next = head;
```

```
        head.next = null;

        //return head;
        return newHead;
    }
}
 经典的题目就是链表逆序啦，一般的链表逆序是让把链表从前到后都逆序，
 这个是给定了起始位置和结束位置，方法是一样的。
 就是维护3个指针，startpoint，node1和node2。
 startpoint永远指向需要开始reverse的点的前一个位置。
 node1指向正序中第一个需要rever的node，node2指向正序中第二个需要reverse的node。
 交换后，node1 在后，node2在前。这样整个链表就逆序好了。
public ListNode reverseBetween(ListNode head, int m, int n) {
        ListNode newhead = new ListNode(-1);
        newhead.next = head;

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

        ListNode startpoint = newhead;//startpoint指向需要开始reverse的前一个
        ListNode node1 = null;//需要reverse到后面去的节点
        ListNode node2 = null;//需要reverse到前面去的节点

        for (int i = 0; i < n; i++) {
            if (i < m-1){
                startpoint = startpoint.next;//找真正的startpoint
            } else if (i == m-1) {//开始第一轮
                node1 = startpoint.next;
                node2 = node1.next;
            }else {
                node1.next = node2.next;//node1交换到node2的后面
                node2.next = startpoint.next;//node2交换到最开始
                startpoint.next = node2;//node2作为新的点
                node2 = node1.next;//node2回归到node1的下一个，继续遍历
            }
        }
        return newhead.next;
    }
}


/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isPalindrome(ListNode head) {
        //do it in O(1) space complexity
        if(head == null || head.next == null)  return true;

        ListNode slow=head, fast =head;
        while(fast!=null && fast.next !=null){
            slow = slow.next;//find the middle point of a linked list
            fast = fast.next.next;
        }

        if(fast!=null){
            //if the length of the linked list is odd
            //jump over the middle point
            slow = slow.next;
        }

        ListNode rhead = reverse(slow);
        ListNode lcur = head, rcur = rhead;
        //because we never wanted to move rhead since we need it be fixed and then recover the list later
        while(rcur !=null){
            if(rcur.val != lcur.val){
                reverse(rhead);//restore
                return false;
            }

            lcur=lcur.next;
            rcur=rcur.next;

        }
        reverse(rhead);//restore the right head
        return true;
```

```
303
304
305     }
306     private ListNode reverse(ListNode n){
307         ListNode pre = null;
308         while(n!=null){
309             //save the next
310             ListNode after = n.next;
311             n.next = pre;//link before
312             //link after
313             //pre.next = n;
314             pre=n;
315
316
317             //update n
318             n=after;
319         }
320         //n=null here and we need to return pre
321         return pre;
322     }
323 }
324 ----------------delete/insert a node -- need dummy and pre/next ---
325 must be careful -- ListNode dummy = new ListNode(-1);
326 dummy.next = head;
327 ListNode scanner = dummy; Then if we move scanner dummy will also be moved?
328 The answer is no, becuase the following dummy is recording head but scanner is
329 searching afterwards
330
331 The following solution is wrong
332 package delete;
333 class ListNode{
334     int val;
335     ListNode next;
336     public ListNode(int val){
337         this.val = val;
338     }
339 }
340 public class Delete {
341     public static void main(String[] args){
342         //3,1,2,3,4,3
343         //delete 3 -> 1,2,4
344         ListNode head = new ListNode(3);
345         ListNode scanner = head;
346         scanner.next = new ListNode(1);
347         scanner = scanner.next;
348         scanner.next = new ListNode(2);
349         scanner = scanner.next;
350         head = helper(head, 3);
351         for(ListNode i = head; i!=null; i=i.next){
352             System.out.println(i.val);
353         }
354     }
355
356     //delete all nodes equaling val in a linked list
357     public static ListNode helper(ListNode head, int val){
358         if(head ==null){
359             return head;
360         }
361
362         ListNode dummy = new ListNode(-1);
363         dummy.next = head;//help to delete the head if needed
364         ListNode cur = head;
365         ListNode pre = dummy;  change pre here afterwards will not impact dummy
366         it is just like two pointers poitning to the same object
367         //ListNode pre = new ListNode(-11);
368         //pre.next = cur;
369         while(cur!=null){
370             if(cur.val == val){
371                 //System.out.println(cur.val);
372                 pre.next = cur.next;
373             }//else{
374                 //search afterwards
375 //              cur = cur.next;
376 //              //can this statement always set pre to be right before the cur?
377 //              pre = pre.next;
378
379                 //order is reversed !!!!!!
380                 //we should do
381                 pre = cur;
382                 cur = cur.next;
```

```
                //similarly!!! if we want an array and use pre and cur
                //we must update pre firstly to be cur
                //then cur=cur.next
                //instead of cur = cur.next; pre.next = cur; this is wrong!!! update pre = cur and cur=cur.next is right!!
                for(int cur=0; cur<s.length(); cur++){
                    //something here
                    pre = cur;
                    //so that next turn cur is +1 but pre is still right before the cur!!!!!
                }
                DUMMY IS ALWAYS -1 AND NOT CHANGED BY THE REFERENCE PRE POINTER
                System.out.println(dummy.val);
            //}
        }
        return dummy.next;
    }
}

The following solution is correct

public class Solution {
    public ListNode removeElements(ListNode head, int val) {
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        ListNode scanner = dummy;
    ///////padding a fake dummy head to remove special case
        while(scanner.next != null){
            if(scanner.next.val == val){
                scanner.next = scanner.next.next;
            }else{
                scanner = scanner.next;
            }
        }
        return dummy.next;
    }
}

-------------------nth node faster/slower------------------------

 这道题也是经典题，利用的是faster和slower双指针来解决。

首先先让faster从起始点往后跑n步。

然后再让slower和faster一起跑，直到faster==null时候，slower所指向的node就是需要删除的节点。

注意，一般链表删除节点时候，需要维护一个prev指针，指向需要删除节点的上一个节点。

为了方便起见，当让slower和faster同时一起跑时，就不让 faster跑null了，
让他停在上一步，faster.next==null时候，这样slower就正好指向要删除节点的上一个节点，
充当了prev指针。这样一来，就很容易做删除操作了。

slower.next = slower.next.next(类似于prev.next = prev.next.next)。

同时，这里还要注意对删除头结点的单独处理，要删除头结点时，没办法帮他维护prev节点，
所以当发现要删除的是头结点时，直接让head = head.next并returnhead就够了。

public static ListNode removeNthFromEnd(ListNode head, int n) {
        if(head == null || head.next == null)
            return null;

        ListNode faster = head;
        ListNode slower = head;

        for(int i = 0; i<n; i++)
            faster = faster.next;

        if(faster == null){
            head = head.next;
            return head;
        }

        while(faster.next != null){
            slower = slower.next;
            faster = faster.next;
        }

        slower.next = slower.next.next;
        return head;

        }
```

```
    }
}

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:
Given 1->2->3->4->5->NULL and k = 2,
return 4->5->1->2->3->NULL.

public class Solution {
    public ListNode rotateRight(ListNode head, int k) {
        ///
        //this is similar to delete the nth node from the end
        //however rotate the array is using bubble sort
        //when n=0  we do not need to make any changes
        if(head ==null|| head.next == null|| k==0){
            return head;//for the list return head means return the list
        }

        //if we need to count the list's length we need to control the head to be unchanged
        ListNode fast = head, slow = head, count = head;
        int len=0;
        while(count !=null){
            count = count.next;
            len++;
        }

        k=k%len;//we need to mod the number to get the real position
        if(k==0){
            return head;
        }

        for(int i =0; i<k; i++){
            //find the nth to the end
            //!!!!!!!!
            fast = fast.next;
        }

        while(fast.next !=null){
            //stop right before the node we want to delete/rotate/break
            fast = fast.next;
            slow = slow.next;
        }
        // head = slow.next;
        // slow.next =null;
        //!!!!!!
        ListNode newhead = slow.next;
        fast.next = head;
        slow.next = null;
        return newhead;
    }
}
http:www.cnblogs.com/EdwardLiu/p/4306556.html
-------------------rotate array---------------------
解法一 [ 时间复杂度O（n），空间复杂度O(1) ]:
以n - k为界，分别对数组的左右两边执行一次逆置；然后对整个数组执行逆置。

reverse(nums, 0, n - k - 1)
reverse(nums, n - k, n - 1)
reverse(nums, 0, n - 1)
Naive想法就是保存一个原数组的拷贝，然后把原数组分成前len-k个元素和后k个元素两部分，
把后k个元素放到前len-k个元素前面去。这样做需要O(N)空间

in-place做法是:
(1) reverse the array;
(2) reverse the first k elements;
(3) reverse the last n-k elements.

The first step moves the first n-k element to the end,
and moves the last k elements to the front.
The next two steps put elements in the right order.

public class Solution {
    public void rotate(int[] nums, int k) {
        int len = nums.length;
        k %= len;
        reverse(nums, 0, len-1);
        reverse(nums, 0, k-1);
        reverse(nums, k, len-1);
    }
```

```
543
544     public void reverse(int[] nums, int l, int r) {
545         while (l <= r) {
546             int temp = nums[l];
547             nums[l] = nums[r];
548             nums[r] = temp;
549             l++;
550             r--;
551         }
552     }
553 }
554
555  (1) reverse the whole array
556  (2) reverse each subarray seperated by ' '
557 public class Solution {
558     public void reverseWords(char[] s) {
559         if (s.length == 0) return;
560         reverse(s, 0, s.length-1);
561         int last = 0;
562         for (int i=0; i<s.length; i++) {
563             if (s[i] == ' ') {
564                 reverse(s, last, i-1);
565                 last = i + 1;
566             }
567         }
568     }
569
570     public void reverse(char[] s, int l, int r) {
571         while (l <= r) {
572             int temp = s[l];
573             s[l] = s[r];
574             s[r] = temp;
575             l++;
576             r--;
577         }
578     }
579 }
580
581 public class Solution {
582     public String reverseWords(String s) {
583         StringBuilder reverse = new StringBuilder();
584         /*
585         Given s = "the sky is blue",
586         return "blue is sky the".
587         */
588         //j record each word's end
589         //and i will record each word's beginning
590         int j = s.length()-1;
591         boolean flag =false;
592         for(int i = s.length()-1; i>=0; i--){
593             if(s.charAt(i-1) == ' '){
594                reverse.append(s.substring(i, j)).append(" ");
595                flag = true;
596                continue;
597             }
598
599             if(s.charAt(i) != ' ' && flag){
600                 //update the end of the word
601                 j = i;
602                 flag = false;
603             }
604
605         }
606         //sb also has the length()
607         return reverse.deleteCharAt(reverse.length()-1).toString();
608     }
609 }
610 ---------------------------------------------------
611 题目大意：
612 编写一个函数删除单链表中（除末尾节点外）的一个节点，只提供待删除节点。
613
614 假如链表是1 -> 2 -> 3 -> 4 给你第3个节点，值为3，则调用你的函数后链表为1 -> 2 -> 4
615
616 解题思路：
617 链表基本操作，记待删除节点为node
618
619 令node.val = node.next.val，node.next = node.next.next即可
620
621 其实简单来说就是把传入节点的后面一个节点的值赋给自己，然后把自己后面的节点删掉即可。
622 public class Solution {
```

```
    public void deleteNode(ListNode node) {
        HANDLING special CASE WHEN THE NODE IS THE LAST ONE
        if(node==null||node.next==null) return;

        node.val = node.next.val;
        node.next = node.next.next;
}
}

----------cycle-------------

public ListNode detectCycle(ListNode head) {
        if(head==null||head.next==null)
            return null;

        ListNode fast = head,slow=head;
        while (true) {
            if (fast == null || fast.next == null) {
            return null;
        }
            slow = slow.next;
            fast = fast.next.next;

            if(fast==slow)
                break;
        }

        slow = head;//slow back to start point
        while(slow != fast){
            slow = slow.next;
            fast = fast.next;
        }
        return slow; //when slow == fast, it is where cycle begins
    }
}

public boolean hasCycle(ListNode head) {
        if(head == null || head.next == null)
            return false;

        ListNode Faster = head, Slower = head;

        while(Faster.next!=null && Faster.next.next!=null){
            Slower = Slower.next;
            Faster = Faster.next.next;

            if(Faster == Slower)
                return true;
        }
        return false;
    }


--------reorder list---------------------------
题解：

题目要重新按照 L0→Ln→L1→Ln-1→L2→Ln-2→…来排列，
看例子1->2->3->4会变成1->4->2->3，拆开来看，是{1，2}和{4，3}的组合，
而{4，3}是{3，4}的逆序。这样问题的解法就出来了。

第一步，将链表分为两部分。
第二步，将第二部分链表逆序。
第三步，将链表重新组合。

public void reorderList(ListNode head) {
        if(head==null||head.next==null)
            return;

        ListNode slow=head, fast=head;
        ListNode firsthalf = head;

        find the middle of the linkedlist
        while(fast.next!=null&&fast.next.next!=null){
            slow = slow.next;
            fast = fast.next.next;
        }

        slow will stop right before the middle node
        and this is easy for us to find 1. the head of second half 2.truncate first half
        ListNode secondhalf = slow.next;
```

```
        slow.next = null;

        reverse secondhalf
        secondhalf = reverseOrder(secondhalf);

        firsthalf 1 2
        secondhalf 4 3
        we want 1 4 2 3

        while (secondhalf != null) {
            temp1 is 2
            temp2 is 3
            ListNode temp1 = firsthalf.next;
            ListNode temp2 = secondhalf.next;

            firsthalf.next = secondhalf;
            secondhalf.next = temp1;

            firsthalf = temp1;
            secondhalf = temp2;
        }

    }
    public static ListNode reverseOrder(ListNode head) {
        if (head == null || head.next == null)
            return head;

        ListNode pre = head;
        ListNode cur = head.next;

        while (curr != null) {
            ListNode after = cur.next;
            cur.next = pre;

            pre = cur;
            cur = after;
        }

        // set head node's next
        head.next = null;

        return pre;
    }
}

----------------------------merge sort----------------------
public class Solution {
        public ListNode sortList(ListNode head) {
            return mergeSort(head);
        }
        private ListNode mergeSort(ListNode head)
        {
            if(head == null || head.next == null)
                return head;
            ListNode walker = head;
            ListNode runner = head;
            while(runner.next!=null && runner.next.next!=null)
            {
                walker = walker.next;
                runner = runner.next.next;
            }
            ListNode head2 = walker.next;
            walker.next = null;
            ListNode head1 = head;
            head1 = mergeSort(head1);
            head2 = mergeSort(head2);
            return merge(head1, head2);
        }
        private ListNode merge(ListNode head1, ListNode head2)
        {
            ListNode helper = new ListNode(0);
            helper.next = head1;
            ListNode pre = helper;
            while(head1!=null && head2!=null)
            {
                if(head1.val<head2.val)
                {
                    head1 = head1.next;
                }
                else
```

```
                {
                    ListNode next = head2.next;
                    head2.next = pre.next;
                    pre.next = head2;
                    head2 = next;
                }
                pre = pre.next;
            }
            if(head2!=null)
            {
                pre.next = head2;
            }
            return helper.next;
        }
}
----------------------------insertion sort----------------------
public class Solution {
    public ListNode insertionSortList(ListNode head) {
            //Insertion Sort就是把一个一个元素往已排好序的list中插入的过程。
            //初始时，sorted list是空，把一个元素插入sorted list中。然后，在每一次插入过程中，都是找到最合适位置进行插入。

            //因为是链表的插入操作，需要维护pre，cur和next3个指针。
            //each turn pre始终指向sorted list的fakehead，cur指向当前需要被插入的元素，next指向下一个需要被插入的元素。
            //because we cannot search from back to front in linkedlist

            //当sortedlist为空以及pre.next所指向的元素比cur指向的元素值要大时，需要把cur元素插入到pre.next所指向元素之前。否则，pre指针后移。最后返回
            if(head == null||head.next == null)  {
                return head;
            }

            //when we do the linked list insertion
            //first we always want a dummy head so that we return dummy.next in the end
            //and we always want the cur/next/pre and cur=head to be the scanner
            ListNode dummy = new ListNode(-1);
            //dummy.next = head;
            ListNode cur = head;//cur is the scanner
            //whenever there is insertion we need to get pre, cur, and next
            while(cur!=null){
                ListNode pre = dummy;//pre is searching for the pre node for insertion
                //next is for the inserted node's next
                ListNode next = cur.next;
                //after each time we reset the postion of pre to be fakehead
                //we need to go through the list to find out the right position to find
                while(pre.next!=null && pre.next.val<cur.val){

                    //attention we need to stop right before the right position
                    //so we need to pre.next.val<cur.val instead of pre.val < cur.val
                    pre = pre.next;
                }
                //we could insert it into the list
                //link the new node with before and after
                cur.next = pre.next;
                pre.next = cur;
                //the next pointer is used to track the position for next's turn
                //the starting point is no longer cur.next since cur is inserted to the new position
                //and therefore we need to update cur to be next which is cur's original cur.next

                cur=next;//next turn

            }
            return dummy.next;
    }
}
--------------------intersection --get length ----------------------

public class Solution {
    public ListNode getIntersectionNode(ListNode headA, ListNode headB) {
        //we need to get the length of the two lists
        //at first
        //and then traverse the longer list by the
        //difference and then move them at the same time
        //until they have the same value
        if(headA == null || headB ==null){
            return null;
        }
        //there is no pointer in java

        //just declare the same kind of type of node
        //to point to the head
//、、ListNode cur = headA;
        int len1 = getlen(headA);
```

```
        int len2= getlen(headB);
        //Math.max / Math.min
        int offset = Math.abs(len1-len2);

        if(len1>len2){
            while(offset >0){
                headA = headA.next;
                offset--;
            }
        }else{
            while(offset>0){
                headB=headB.next;
                offset--;
            }
        }

        //now headA and headB's rest of lists have the
        //same length!!!
        while(headA!=null){
            if(headA == headB){
                //we just need to return one of these two heads because they are identical
                return headA;
            }

            //we need to carry on
            headA = headA.next;
            headB = headB.next;

        }

        return null;
    }
    public int getlen(ListNode head){
        //because we are passing by value and therefore
        //we could just passing in the head of each linkedlist
        //while we do not need to worry about the head was gone during the searching
        int len=0;
        while(head!=null){
            head = head.next;
            len++;
        }
        return len;
    }
}

----------------由于要删除节点需要使用被删节点的前节点。所以实际写的时候考察的是p->next->val和x的比较。------
public class Solution {
    public ListNode partition(ListNode head, int x) {
        //这类头节点经常要插入、删除的题目，第一反应就是试试使用dummy头节点来简化代码。
        //由于不要求sort，只要求partition。可以建立一个新的list l2。遍历原list l1的每个节点p。
//          // p->val < x，保留。
//         // p->val >= x，从l1中移出并插入l2。
//         //由于要删除节点需要使用被删节点的前节点。所以实际写的时候考察的是p->next->val和x的比较。
        if(head == null ||head.next == null){
            return head;
        }

        ListNode dummy = new ListNode(-1);
        ListNode dummy_scanner = dummy;

        //dummy is for creating the new list
        //and we need to creat the new list and return it
        //we need to scan the first head of original list too
        ListNode original = new ListNode(-1);
        original.next = head;
        ListNode scanner = original;
        while(scanner.next!=null){
            if(scanner.next.val < x){
                //reserve in the original list
                scanner = scanner.next;
            }else{
                //1.delete it from the original list
                //2.append it to the tail of the new list
                //when we need to delete a node
                //we must know the pre!!!! for the node!!!! to delete it!!!!

                //append to new list
                dummy_scanner.next = scanner.next;
                //delete from old list
                scanner.next = scanner.next.next;
```

```
                dummy_scanner = dummy_scanner.next;

            }
        }
        // 最后，把小链表接在大链表上，别忘了把大链表的结尾赋成null。*/
        dummy_scanner.next = null;//terminate it

        //all small ones in the old list
        //large ones in the new list
        //connect together
        scanner.next = dummy.next;
        return original.next;
}
------------------remove dups-------------------
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if(head == null || head.next == null){
            return head;
        }
        //once mentioned hash set we need to use data structure like hashset to avoid dups

        HashSet<Integer> unique = new HashSet<Integer>();
        unique.add(head.val);
        ListNode cur;
        //cur is to check value
        ListNode pre = head;
        //pre is also the scanner and also the previous node for the current node
        //once find dups we need to pre.next = cur.next
        //otherwise just do pre = pre.next
        for( cur = head.next ; cur!=null; cur = cur.next){
            System.out.print("a");
            if(!unique.contains(cur.val)){
                System.out.print("asda" + cur.val);
                //this is a unque value
                unique.add(cur.val);
                pre = pre.next;
            }else{
                System.out.print("121");
                pre.next = cur.next;
            }
            //originally
            //I wrote pre = pre.next; here
            //but the thing is that we should not carry on the pre when there is a duplicate afterwards
        }
        return head;
    }
}

Given 1->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        //whenever we meet with dups we need to use hash set to help us solve the problem
        if(head == null){
            return head;
        }
        //why we need dummy again!!!!!
        ListNode dummy = new ListNode(0);
        dummy.next = head;
        head = dummy;

        //as we already have dummy to record the *head* which is dummy.next
        //the head could actually work as the scanner

        while (head.next != null && head.next.next != null) {
            //we need to peek at the next and the next/next for dummy
            //because now head is pointing to the dummy
            if (head.next.val == head.next.next.val) {
                int dup = head.next.val;
                while(head.next!=null && head.next.val == dup){
                    //once we found one dup
                    //we could jump over all these dups
                    head.next = head.next.next;
                }
            }else{

                //else we need to move the scanner forward
                //to check next pait of nodes
                head = head.next;
            }
        }
```

```
1023
1024            return dummy.next;//because head is no longer head
1025        }
1026    }
1027
1028    public class Solution {
1029        public int removeDuplicates(int[] nums) {
1030            //count for the corner case at first
1031            if(nums.length <2){
1032                return nums.length;
1033            }
1034            //two pointers
1035            int index=0;
1036            for(int i =1; i<nums.length ; i++){
1037                if(nums[i] != nums[index]){
1038                    //just move it to there
1039                    nums[++index] = nums[i];
1040                }
1041            }
1042            ///we are return the length of the whole array!!!!!!
1043            //therefore index and the length have the relationship of
1044            //length = index +1
1045            return index +1;
1046        }
1047    }
1048
1049    public class Solution {
1050        public int removeDuplicates(int[] nums) {
1051            if(nums == null || nums.length == 0){
1052                return 0;
1053            }
1054            //two pointers!
1055            if(nums.length == 1 || nums.length == 2){
1056                return nums.length == 1 ? 1:2;
1057            }
1058            //because the array has already been sorted
1059            int index = 1;//two pointers
1060            for(int i=2; i<nums.length; i++){
1061                //not care about the rep
1062                //we just move the non-rep to here and return index which is the new length
1063                if(nums[index-1] != nums[i]){
1064                    //according to the characteristic of sorted array
1065                    //if nums[index] == nums[index-1] == nums[i]
1066                    //we need to move
1067                    //
1068                    //but nums[index] must equal nums[index-1] if there is dup
1069                    nums[++index] = nums[i];
1070                }
1071            }
1072            return index+1;//index to length
1073        }
1074    }
1075    -----------------merge lists------------------
1076
1077    public class Solution {
1078        public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
1079            //we need to think about if one list is longer than the other
1080
1081            if(l1 == null){
1082                return l2;
1083            }
1084
1085            if(l2 == null){
1086                return l1;
1087            }
1088
1089            ListNode dummy = new ListNode(0);
1090            //does this means they both have the same position?
1091            //i WANT TO KEEP HEAD but still want to add things later
1092            //does this just copy by value? or just copy by address?
1093
1094            /*Java does manipulate objects by reference, and all object variables are references. However, Java doesn't pass method arguments b
1095            */
1096
1097            //here similarly scanner is just a pointer --> like ListNode head = new ListNode() which means you are assign the object to the poi
1098            //so scanner is pointing at the same address with head
1099            ListNode scanner = dummy;
1100            //head.val = l1.val;
1101            //while(l1.next != null && l2.next !=null){
1102            while(l1 != null && l2 !=null){
```

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

                scanner.next = l1;
                l1 = l1.next;
                scanner = scanner.next;
            }else{
                scanner.next = l2;
                l2 = l2.next;
                scanner = scanner.next;
            }
        }
     //this is wrong!!!!!!!! because if 1
     //2
     //then l1.next always is null
     //l2.next always is null
     //cannot pass the test
     //!!!!
        //if(l1.next != null)
          if(l1 != null){
              scanner.next = l1;
          }
          if(l2 != null)
          {
              scanner.next = l2;
          }
        // while(l1.next !=null){
        //     scanner.next = l1;
        //         l1 = l1.next;
        //         scanner = scanner.next;
        // }
        return dummy.next;
    }
}
public class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if(lists.length ==0 || lists == null){
            return null;
        }

        Comparator<ListNode> cmp = new Comparator<ListNode>(){
            public int compare(ListNode o1, ListNode o2){
                return o1.val-o2.val;//ascending order
            }
        };

        PriorityQueue<ListNode> q = new PriorityQueue<>(lists.length, cmp);

        for(ListNode head : lists){
           if(head!=null){
                q.offer(head);
           }
        }

        ListNode dummy = new ListNode(-1);//easy to insert and add to the final result list
        ListNode scanner = dummy;

        while(!q.isEmpty()){
            int size = q.size();
            //here we are using BFS similar to the level order traversal in the tree
            //actually here we do not need to follow certain pattern
            //we could delete for!!!
            //because the priorityQueue<>(size,cmp) is already sorted by cmp
            //anytime we insert or add a new value into it -- it will be automatically sorted again
            for(int i=0; i< size; i++){
                ListNode out = q.poll();
                scanner.next = out;
                if(out.next !=null){
                    q.offer(out.next);
                }

                scanner = scanner.next;
            }
        }

        return dummy.next;

    }
}
```