>

[Java]Leetcode236 Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”

```
    _______3_____
   /             \
  ___5__          __1__
 /      \        /     \
 6      _2       0      8
       / \
       7  4
```

For example, the lowest common ancestor (LCA) of nodes `5` and `1` is `3`. Another example is LCA of nodes `5` and `4` is `5`, since a node can be a descendant of itself according to the LCA definition.

题意：之前一题的树是二叉查找树，而本题的树就是一棵普通的二叉树。给定两个节点，判断这两个节点的相

解题思路：先遍历左子树，返回匹配的点，没有返回null。后遍历右子树，返回匹配的点，没有返回null。如果

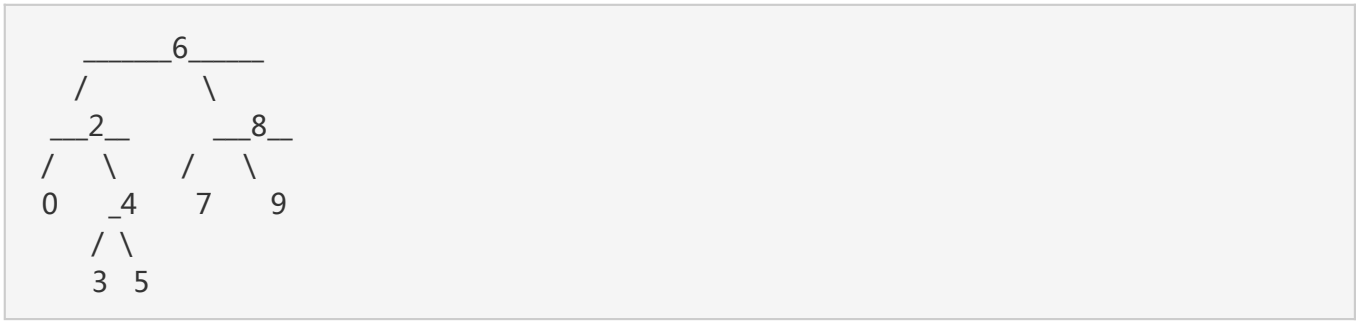
```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if(root==null)return null;
        if(root==p||root==q)return root;
        TreeNode left=lowestCommonAncestor(root.left,p,q);
        TreeNode right=lowestCommonAncestor(root.right,p,q);
        if(left!=null&&right!=null)return root;
        return left!=null?left:right;
    }
```


[Java]LeetCode235 Lowest Common Ancestor of a Binary Search Tree

Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes

in the BST.

According to the definition of LCA on Wikipedia: “The lowest common ancestor is defined between two nodes v and w as the lowest node in T that has both v and w as descendants (where we allow a node to be a descendant of itself).”



For example, the lowest common ancestor (LCA) of nodes 2 and 8 is 6. Another example is LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself according to the LCA definition.

题意，给出两个节点，让你判断两个节点的父节点，两个节点中也有可能有父节点。而且这题的树是二叉排序

解题思路：1）首先判断两个节点中是否有根节点，如果有，返回根节点。2）如果没有，比较root.val>Math

```
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if(root==p||root==q)return root;
    if(root.val>Math.max(p.val,q.val))
    {
        root=lowestCommonAncestor(root.left,p,q);
    }else if(root.val<Math.min(p.val,q.val))
    {
        root=lowestCommonAncestor(root.right,p,q);
    }
    return root;
}
```


[Java]LeetCode57 Insert Interval

Given a set of non-overlapping intervals, insert a new interval into the intervals (merge if necessary).

You may assume that the intervals were initially sorted according to their start

times.

Example 1:
Given intervals [1,3],[6,9], insert and merge [2,5] in as [1,5],[6,9].

Example 2:
Given [1,2],[3,5],[6,7],[8,10],[12,16], insert and merge [4,9] in as [1,2],[3,10],[12,16].

This is because the new interval [4,9] overlaps with [3,5],[6,7],[8,10].
题意：该题与56题很类似，不过题意给出每个区间的start是递增的，所以我们不需要排序。该题需要我们添加一个区间，然后进行融合。这题在56题的基础上增加了区间判断的复杂度。
包含如下如所示的六种情况。

而这六种情况又可以合并成三种解决方式。看如下代码：

```
/**
* Definition for an interval.
* public class Interval {
*     int start;
*     int end;
*     Interval() { start = 0; end = 0; }
*     Interval(int s, int e) { start = s; end = e; }
* }
*/
public class Solution {
   public List<Interval> insert(List<Interval> intervals, Interval newInterval) {
      // 先判断newInterval是否在intervals的范围内
       if (newInterval == null)
           return intervals;
       int len = intervals.size();
       if (len == 0)
       {
           intervals.add(newInterval);
           return intervals;
       }
       List<Interval> res=new ArrayList<Interval>();
       for(Interval interval:intervals)
       {
           if(interval.end<newInterval.start)//newInterval在中间的情况
           {
               res.add(interval);
           }else if(interval.start>newInterval.end)//newInterval插入最前端的情况
           {
               res.add(newInterval);
               newInterval=interval;//这个地方很重要，就是找到了待插入区间位置，指定新的newInterval，因为
           }else if(interval.start<=newInterval.end||interval.end>=newInterval.start)//有重合部分的四种情况
           {
               newInterval=new Interval(Math.min(interval.start,newInterval.start),Math.max(interval.end,ne
           }
       }
```

```
        res.add(newInterval);
        return res;
    }
}
```


[Java]LeetCode53 Maximum Subarray

Find the contiguous subarray within an array (containing at least one number) which has the largest sum.

For example, given the array [?2,1,?3,4,?1,2,1,?5,4],

the contiguous subarray [4,?1,2,1] has the largest sum = 6.

题意：就是找出数组中最大值的字串。这一题不是很难，关键在于分析。我们发现如果有正数，那最大值的子串肯定是从正数开始，但如果全是负数的话，那最大值肯定是最大负数。

第一版代码：这是第一版代码，逻辑不是很紧密，代码也很冗余，但是大致的思想是这样的。

```
public int maxSubArray(int[] nums) {
        int len=nums.length;
        if(len==0)return 0;
        int sum=nums[0];
        int max=sum;
        for(int i=1;i<len;i++)
        {
            if(sum>=0)
            {
             //若此时的nums[i]<0
             if(nums[i]<0)
             {
                 if(max<sum)max=sum;//记录前边的已经遍历过的最大值
             }
             sum+=nums[i];
            }else
            {
                //sum<=0
                if(max<nums[i])max=nums[i];//全是负数的情况下，找出最大值。
                sum=nums[i];
            }
        }
      return max>sum?max:sum;
    }
```

第二版代码：

```
public int maxSubArray(int[] nums) {
      int len=nums.length;
      if(len==0)return 0;
      int sum=nums[0];
      int max=sum;
      for(int i=1;i<len;i++)
      {
          if(sum>=0)//正数进行累加
            sum+=nums[i];
          else//如果是负数的话，就不要累加了
            sum=nums[i];
          max=Math.max(max,sum);
      }
     return max;
  }
```


[MySQL]LeetCode196 Delete Duplicate Emails

Write a SQL query to delete all duplicate email entries in a table named `Person`, keeping only unique emails based on its smallest Id.

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
| 3  | john@example.com |
+----+------------------+
Id is the primary key column for this table.
```

For example, after running your query, the above `Person` table should have the following rows:

```
+----+------------------+
| Id | Email            |
+----+------------------+
| 1  | john@example.com |
| 2  | bob@example.com  |
+----+------------------+
```

题意:将一个表中重复的email行去掉只保留一行，且保留的Id最小。

解法一：

```
delete from Person where Id not in
(select min_id from
(select min(id) as min_id from Person group by Email)
as tmp);
```

解法二：

```
delete p1 from Person p1 inner join Person p2 where p1.Email=p2.Email and p1.Id>p2.Id  //内连接将
```

有人看了上面解法一的答案：会问select min_id from这句话好像没啥用。得到的不还是内层select的结果么

You can't specify target table 'Person' for update in FROM clause 意思是，你不能在from的子句

[Java]Leetcode69 Sqrt(x)

Implement `int sqrt(int x)`.Compute and return the square root of x.

题意：算一个数的平方根。

好不容易想到用二分查找来解决改题，但是也破费了点力气，放到eclipse中调试来出来。注意点就是：取中值相乘，有可能会超过整数的最大范围，所以比较的时候就会出错。所以在定义的时候全部定义为long型。

第一版：

```
public int mySqrt(int x)
{
 if (x == 0)
  return 0;
 long low = 1;
 long high = x;
 long tmp;
 long mid = 1;
 while (low <= high)//二分查找
 {
  mid = (low + high) / 2;
  tmp = mid * mid;
  if (tmp == x)
   return (int)mid;
  else if (tmp > x)
   high = mid - 1;
  else if (tmp < x)
   low = mid + 1;
```

```
 }
 return (int)((mid*mid)>x?mid-1:mid);
}
```

我们可以将上述的代码再改进一下：

```
public int mySqrt(int x)
{
 long low = 0;
 long high = x/2+1;//平方根的值按规律发现不会大于它的中值+1。这样每个查找就少了一次
 long tmp;
 long mid = 1;
 while (low <= high)
 {
  mid = (low + high) / 2;
  tmp = mid * mid;
  if (tmp == x)
   return (int)mid;
  else if (tmp > x)
   high = mid - 1;
  else if (tmp < x)
   low = mid + 1;
 }
 return (int)high;
}
```


LeetCode88 Merge Sorted Array

Given two sorted integer arrays nums1 and nums2, merge nums2 into nums1 as one sorted array.

Note:

You may assume that nums1 has enough space (size that is greater or equal to m + n) to hold additional elements from nums2. The number of elements initialized in nums1and nums2 are m and n respectively.

题意：给定两个有序数组，然后将数组2融合到数组1中。

这道题还是很简单的，假设两个数组给定的顺序都是升序的。这题的方法有很多，在选择方法的时候，我们知道，数组中插入新的数据，就会涉及到数组元素的移动，那用什么方法能够尽可能的减少不必要的移动呢。或者是数字插入后每位数字只需要移动一次即是它最终的位置。

思路是：从后往前比较，将数组1和数组2中的数字从后往前比较，谁大就移到后边，直到数组1或数组2中的数字比较完。然后再做剩余部分的处理。

程序如下：

```
public void merge(int[] nums1, int m, int[] nums2, int n) {
      if(n==0)return;
      //将nums2中的数从后往前比
      int j=m-1;
      int i=n-1;
      while(i>=0&&j>=0)
      {
        if(nums2[i]>nums1[j])
          {
             nums1[j+i+1]=nums2[i];
             i--;
          }else
          {
             nums1[j+i+1]=nums1[j];
             j--;
          }

      }
     if(i>=0)
     {
        for(int k=0;k<=i;k++)
        {
           nums1[k]=nums2[k];
        }
     }
  }
```


[Java]LeetCode5 Longest Palindromic Substring

Given a string S, find the longest palindromic substring in S. You may assume that the maximum length of S is 1000, and there exists one unique longest palindromic substring.

题意：求字符串中最长的回文

感觉这题不难，可以这样想，设置两个指针，分别对应0，len-1。

比如从第一个字符开始，abababac，我们可以找a出现的位置，然后以第一个a与每个a作为一段。判断每一段是否是回文，因为是从后往前查找的，所以第一个找到的肯定是这一次查找中最长的回文，然后一次第2个字符，第3个字符。这个想法在字符串中的字符不一样的时候比较好。但是如果字符中出现的例如aaaaaabcaaaaa这样的话，就不佳了。在leetcode中一测试,果然Time Limited。

第二个想法，算是逆推法，既然求的是回文，回文的字符是对称的。我们可以先确定回文的中心，回文的中心，奇数的时候是一个，偶数的时候是两个。分别向中心的两边进行匹配，判断是否相等，从而计算回文的长度。

代码如下：

```
public static String longestPalindrome(String s) {
      if(s.isEmpty()||s.length()==1)return s;
      String longest=s.substring(0,1);
      for(int i=0;i<s.length();i++)
      {
         //将i作为中心,获取最长的字符串
         String tmp=isPalindrome(s,i,i);
         if(tmp.length()>longest.length())longest=tmp;
         //将i,i+1作为中心，获得最长的字符串
          tmp=isPalindrome(s,i,i+1);
         if(tmp.length()>longest.length())longest=tmp;
      }
      return longest;
  }
private static String isPalindrome(String s, int start, int end)
   {
       // TODO Auto-generated method stub
       while(start>=0&&end<=s.length()-1&&s.charAt(start)==s.charAt(end))
       {
          start--;
          end++;
       }
       return s.substring(start+1,end);
   }
```


[JAVA]LeetCode93 Restore IP Addresses

Given a string containing only digits, restore it by returning all possible valid IP address combinations.

For example:
Given "25525511135",

return ["255.255.11.135", "255.255.111.35"]. (Order does not matter)

题意是：给定一个字符串，字符串全由数字组成，在判断合法性的同时，输出所有正确的IP组合数。注意，当某个区段0开头时，那该区段只能是0

解题思路，很明显的动态规划的思想解题。就是先求出一个区段的可能的组合数，然后在第一个区段的基础上，求第二个区段的可能组合数，以此类推，因为每个区段的可能有很多，也就是剩余数字的个数很根据第一个区段的个数的变化而变化。所以注意处理这个变化，再加上细节的处理，想想还是可以做出来的。

解题第一版：使用的迭代方式求解的。虽然代码有点冗余，基本上就是这个思路。

```
public static List<String> restoreIpAddresses(String s) {
        List<String> res=new ArrayList<String>();
        if(s==null)return res;
        int len=s.length();
        if(len<4||len>12)return res;
        int area=4;
        int i=0;
        while(area>0)
        {
        res=mergeString(s,res,len,area);//area代表目前求解的是第几个区段的IP,4代表的是第1个区段，3代表的
        area--;
        }
       return res;
    }
    static List<String> mergeString(String s,List<String> list,int len,int area)
    {
        int i=1;
        int listSize=list.size();
        List<String> tmpList=new ArrayList<String>();
        StringBuffer buffer=new StringBuffer();
        char ch=' ';
        int tmp=0;
        if(listSize==0)//求第一个区段的IP可能组合。
        {
          while(i<=3&&(len-i)>=1*(area-1))
          {
            ch=s.charAt(i-1);
            if(ch=='0'&&i==1)//当第一区段的第一位值为0的时候，该区段有且只有一种情况，那就是0
             {
             if((len-i)<=3*(area-1))//只有剩余的数字个数小于3*3,区段为0的数才算合法的，加入tmpList中
             {
             tmpList.add(String.valueOf(ch));
             }
             break;
             }
            buffer.append(ch);
            tmp=Integer.valueOf(buffer.toString());
            if(tmp<=255)//所求的数不能大于255
            {
            if((len-i)<=3*(area-1))//剩余的个数不能小于3*3
            tmpList.add(buffer.toString());
            }else break;
            i++;
```

```
        }
      }else//求剩下区段的可能组合
      {
        for(int j=0;j<listSize;j++)//数组中前面字段的所有组合，进行单独组合求解可能数
        {
          i=1;
          String tmpStr=list.get(j);
          int strLen=tmpStr.length()-(3-area);//3-area是减去小数点的个数，就是真实字符串的长度
          String str="";
          while(i<=3&&(len-i-strLen)>=1*(area-1))//i代表的是每个区段的最常长度只能是3以及剩余长度应i
          {                                    //比如剩余区段是2，那最小长度应大于2，最大长度应大于2*3
           buffer=buffer.delete(0,buffer.length());
           buffer.append(tmpStr);
              buffer.append(".");
              ch=s.charAt(strLen+i-1);
              if(ch=='0'&&i==1)
              {
             buffer.append(ch);
             if((len-i-strLen)<=3*(area-1))
             {
                tmpList.add(buffer.toString());
             }
             break;
              }
              str+=ch;
              tmp=Integer.valueOf(str);
             if(tmp<=255)
             {
              buffer.append(str);
              if((len-i-strLen)<=3*(area-1))
              tmpList.add(buffer.toString());
             }else
                 break;
             i++;
            }
          }
        }
       return tmpList;
      }
```

第二版，使用递归来解题

```
 public ArrayList<String> restoreIpAddresses(String s) {
      ArrayList<String> res = new ArrayList<String>();
      if(s==null)return res;
      int len=s.length();
      if (len<4||len>12) return res;
      String str="";
      mergeString(s,str,res,0);
      return res;
  }

  public void mergeString(String s, String str, ArrayList<String> res, int area){
      if (area == 3 && isValid(s)) {
          res.add(str + s);
          return;
      }
```

```
        for(int i=1; i<=3 && i<s.length(); i++){
            String substr = s.substring(0,i);
            if (isValid(substr)){
                mergeString(s.substring(i),str + substr + '.', res, area+1);
            }
        }
    }

    public boolean isValid(String s){
        if (s.charAt(0)=='0') return s.equals("0");
        int num = Integer.parseInt(s);
        return num<=255 && num>0;
    }
}
```


[Java] LeetCode32 Longest Valid Parentheses

Given a string containing just the characters '(' and ')', find the length of the longest valid (well-formed) parentheses substring.

For "(()", the longest valid parentheses substring is "()", which has length = 2.

Another example is ")()())", where the longest valid parentheses substring is "()()", which has length = 4.

题意：给定一串包含括号的字符串，求字长有效的括号串。能顺序匹配的。这一题刚开始想好久~~，用stack来做题，很容易求出字符串中所有有效的括号长度。但是如何求有效子串呢？如果我们能找出无效括号的index，将有效的index减去上一个无效的index，那么就是该字符的有效子串。弄清楚这一点求最大的有效字符串也会变得很容易。

```
public int longestValidParentheses(String s) {
    if(s==null)return 0;
    int len=s.length();
    int i=0;
    Stack<Integer> stack=new Stack<Integer>();
    char ch;
    int res=0;
    while(i<len)
    {
        ch=s.charAt(i);
        if(ch=='(')
        stack.push(i);//我们变换思路，将括号的index入栈
        else
        {
            if(!stack.isEmpty()&&s.charAt(stack.peek())=='(')//如果是’)'，且与stack顶括号匹配时，弹出
            {
                stack.pop();
                res=Math.max(stack.isEmpty()?i+1:i-stack.peek(),res);//为空的话，证明前面没有无效括号，将i
            }else
```

```
            {
                stack.push(i);
            }
        }
        i++;
    }
    return res;
}
```


[Java]LeetCode17 Letter Combinations of a Phone Number

Given a digit string, return all possible letter combinations that the number could represent.
A mapping of digit to letters (just like on the telephone buttons) is given below.

```
Input:Digit string "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"].
```

题意：一个有数字组成的字符串，按照手机上数字所对应的字符串，输出所有可以出现的组合。字符串中的顺

解题思路：这是动态累加的程序。如果说，"23",我先求出2的组合，然后将2的组合与3中所对应的所有字符一

```
public static List<String> letterCombinations(String digits) {
        List<String> list=new ArrayList<String>();
        List<String> res=new ArrayList<String>();
        int len=digits.length();
        if(len==0)return res;
        char ch;
        String tmp=null;
        for(int i=0;i<len;i++)
        {
           ch=digits.charAt(i);
           if(ch!='0'&&ch!='1')
           {
            tmp=digit2String(ch);
            list.add(tmp);
           }
        }
        len=list.size();
```

```
        for(int i=0;i<len;i++)
        {
            res=mergeListAndString(res,list.get(i));
        }
        return res;
    }
    static List<String> mergeListAndString(List<String> list,String str)
    {
     if(str==null)return list;
        List<String> resTmp=new ArrayList<String>();
        if(list.size()==0)
        {
            for(int i=0;i<str.length();i++)
            list.add(String.valueOf(str.charAt(i)));
            return list;
        }else
        {
           for(int j=0;j<str.length();j++)
           {
               for(int k=0;k<list.size();k++)
               {
                resTmp.add(str.charAt(j)+list.get(k));
               }
           }
           return resTmp;
        }
    }
    static String digit2String(char ch)
    {
        String str=null;
        switch(ch)
        {
            case '2':str="abc";break;
            case '3':str="def";break;
            case '4':str="ghi";break;
            case '5':str="jkl";break;
            case '6':str="mno";break;
            case '7':str="pqrs";break;
            case '8':str="tuv";break;
            case '9':str="wxyz";break;
            default: str=null;break;
        }
        return str;
    }
```

这是第一版代码，虽然通过了。但是代码很长。接下来，看看有没有办法精简一下。

```
public static List<String> letterCombinations(String digits) {
            String[] digits2String={"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};
        List<String> res=new ArrayList<String>();
        int len=digits.length();
```

```
        if(len==0)return res;
        char ch;
        String tmp=null;
        for(int i=0;i<len;i++)
        {
           ch=digits.charAt(i);
           if(ch<='9'&&ch>'1')
           {
            tmp=digits2String[ch-'0'];
            res=mergeListAndString(res,tmp);
           }
        }
        return res;
    }
    static List<String> mergeListAndString(List<String> list,String str)
    {
     if(str==null)return list;
        List<String> resTmp=new ArrayList<String>();
        if(list.size()==0)
        {
            for(int i=0;i<str.length();i++)
            list.add(String.valueOf(str.charAt(i)));
            return list;
        }else
        {
           for(int j=0;j<list.size();j++)
           {
               for(int k=0;k<str.length();k++)
               {
                resTmp.add(list.get(j)+str.charAt(k));
               }
           }
           return resTmp;
        }
    }
```


[Java]LeetCode22 Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

For example, given n = 3, a solution set is:

"((()))", "(()())", "(())()", "()(())", "()()()"

题意：给出括号的对数，输出格式正确的括号排列。就是正常闭合的括号。

解题思路：用递归的方法。

```
public List<String> generateParenthesis(int n) {
    List<String> result=new ArrayList<String>();
    if(n==0)return result;
    StringBuffer buffer=new StringBuffer();
    addParenthesis(buffer,result,n,n);
    return result;
   }
   void addParenthesis(StringBuffer buffer,List<String> result,int leftNum,int rightNum)
   {
       if(leftNum>rightNum)return;//如果左括号的个数大于右括号的，返回。
       if(leftNum==0&&rightNum==0)
       {
           result.add(buffer.toString());
       }
       if(leftNum>0)//加左括号
       {
           buffer.append('(');
           addParenthesis(buffer,result,leftNum-1,rightNum);
           buffer.delete(buffer.length()-1, buffer.length());

       }
       if(rightNum>0)//加右括号
       {
           buffer.append(')');
           addParenthesis(buffer,result,leftNum,rightNum-1);
           buffer.delete(buffer.length()-1, buffer.length());

       }
   }
```


[Java]LeetCode49 Anagrams

Given an array of strings, return all groups of strings that are anagrams.

Note: All inputs will be in lower-case.

题意，就是判断字符数组中是否有回文字符串（判断两个字符串是否由相同的字母组成，但是顺序有可能不一样）。

解题思路：首先想到的是将数组中的字符按字母顺序重排列。然后用map存储，字符串做为key值，用一个List链表记录排序之前的字符串在数组中的index。然后判读list中长度，大于1的说明是有回文字符，加入list。

```
public List<String> anagrams(String[] strs) {
        List<String> list=new ArrayList<String>();
        int len=strs.length;
        if(len<=1)return list;
        Map<String,List<Integer>> map=new HashMap<String,List<Integer>>();
        String str=null;
        StringBuffer buffer=new StringBuffer();
        for(int i=0;i<len;i++)
        {
         str=strs[i];
            char[] chars=str.toCharArray();
            Arrays.sort(chars);
            for(char ch:chars)
            {
                buffer.append(ch);
            }
            str=buffer.toString();
            buffer.delete(0,str.length());
            if(map.get(str)==null)
            {
             List<Integer> tmpList=new ArrayList<Integer>();
             tmpList.add(i);
             map.put(str, tmpList);
            }
            else
            {
             map.get(str).add(i);
            }
        }
        Set<String> keySet=map.keySet();
        for(Iterator<String> iter=keySet.iterator();iter.hasNext();)
        {
            String keyStr=iter.next();
            List<Integer> listIndex=map.get(keyStr);
            len=listIndex.size();
            if(len<=1)continue;
            for(int i=0;i<len;i++)
            {
            list.add(strs[listIndex.get(i)]);
            }
        }
     return list;
    }
```

运行的时间比较长，766ms。是否能优化一下呢。在判断的时候直接存储。

```
public List<String> anagrams(String[] strs) {
        List<String> list=new ArrayList<String>();
        int len=strs.length;
        if(len<=1)return list;
        Map<String,Integer> map=new HashMap<String,Integer>();
        String str=null;
        for(int i=0;i<len;i++)
```

```
    {
       str=strs[i];
        char[] chars=str.toCharArray();
        Arrays.sort(chars);
        str=new String(chars);
        if(map.containsKey(str))//判断map中是否有重排列的str
        {
            int index=map.get(str);
            if(index!=-1)//判断是否是第二次存在相同的key。如果是，将第一次的index所对应的数组值取出加
            {
                list.add(strs[index]);
                map.put(str,-1);
            }
            list.add(strs[i]);
        }else
        {
            map.put(str,i);
        }
    }
 return list;
}
```


[Java]Leetcode14 Longest Common Prefix

Write a function to find the longest common prefix string amongst an array of strings.

就是一个字符数组，找字符数组中所有字符串中的最常前缀。

解题思路：首先两个字符比较，找出最长前缀，然后拿前缀与其他字符比较。这里面要注意的是，比较的次数最多是最短字符的长度。

```
public String longestCommonPrefix(List<String> strs) {
    int strsLen=strs.size();
    if(strsLen==0)return "";
    int i=0;
    String str=null;
    String longestCommonPrefix="";
    while(i<strsLen)
    {
       str=strs.get(i);
       if(i==0)longestCommonPrefix=str;
       else
       longestCommonPrefix=LCP2String(str,longestCommonPrefix);
       i++;
```

```
        }
        return longestCommonPrefix;
    }
    public String LCP2String(String str1,String str2)//比较两个字符的最长前缀
    {
        int len1=str1.length();
        int len2=str2.length();
        int len=len1>len2?len2:len1;
        int i=0;
        char ch1=' ';
        char ch2=' ';
        while(i<len)
        {
           ch1=str1.charAt(i);
           ch2=str2.charAt(i);
           if(ch1==ch2)i++;
           else break;
        }
        if(i==0)return "";
        else return str1.substring(0,i);
    }
```


[Java]Leetcode165 Compare Version Numbers

Compare two version numbers version1 and version2.
If version1 > version2 return 1, if version1 < version2 return -1, otherwise return 0.

You may assume that the version strings are non-empty and contain only digits and the . character.
The . character does not represent a decimal point and is used to separate number sequences.
For instance, 2.5 is not "two and a half" or "half way to version three", it is the fifth second-level revision of the second first-level revision.

Here is an example of version numbers ordering:

```
0.1 < 1.1 < 1.2 < 13.37
```

题意是：比较版本号的大小，假设版本号只含有数字和小数点。这一题首先可能想到的是字符转换为整型比较大小，但是版本号中可以有多个小数点，这一点就不符合了。于是想到用split()方法，以小数点为分界符，分成一个字符数组，再一一比较。

代码如下：

```
public int compareVersion(String version1, String version2) {
    String[] ver1=version1.split("\\.");
    String[] ver2=version2.split("\\.");
    int len1=ver1.length;
    int len2=ver2.length;
    int len=0;
    if(len2<=len1)len=len2;
    else len=len1;
    int num1=0;
    int num2=0;
    int i=0;
    while(i<len)
    {
      num1=Integer.valueOf(ver1[i]);
      num2=Integer.valueOf(ver2[i]);
      if(num1==num2)i++;
      else if(num1<num2)return -1;
      else return 1;
    }
    while(len1>i)//有小数点，但是小数点后的值为0的情况，不为0返回1，都为0就return 0.
    {
     if(Integer.valueOf(ver1[i++])!=0)return 1;
    }
    i=len;
    while(len2>i)
    {
      if(Integer.valueOf(ver2[i++])!=0)return -1;
    }
    return 0;
  }
```


[JAVA]LeetCode8 String to Integer (atoi)

Implement atoi to convert a string to an integer.

Hint: Carefully consider all possible input cases. If you want a challenge, please do not see below and ask yourself what are the possible input cases.

Notes: It is intended for this problem to be specified vaguely (ie, no given input specs). You are responsible to gather all the input requirements up front.

题意：将字符串转换为整型。这一题感觉不是很简单，经过多次修改，终于accepted!!!

考虑的问题比较多，首先字符串转换中遇见非数字字符的处理，字符串超出最大整型和最小整型怎么处理。

开题遇到空格，用trim()去空格。

+，-号只可能在去空格后第一位。

中间遇到非数字字符，直接返回前一段字符，判断是否有效，并转换为整型。

代码如下：

```
public int myAtoi(String str) {
    if(str==null)return 0;
    str=str.trim();//去空格
    int len=str.length();
    if(len==0)return 0;
    char signal='+';
    int singalNum=0;
    int i=0;
    char ch=str.charAt(i);
    double result=0;
    while(i<len)
    {
        ch=str.charAt(i);
        if(ch=='-'||ch=='+')//判断符号
        {
            if(i!=0)return 0;//如果符号不在首位，返回0
            if(ch=='-')signal='-';
        }else if(ch>='0'&&ch<='9')
        {
            result=result*10+(str.charAt(i)-'0');//处理数字
        }else if(ch<'0'||ch>'9')//遇到非数字字符，跳出循环，只计算非数字之前的合法数字字符
        {
            break;
        }
        i++;
    }
    if(signal=='-')
    result=-1*result;
    if(result>Integer.MAX_VALUE)return Integer.MAX_VALUE;
    if(result<Integer.MIN_VALUE)return Integer.MIN_VALUE;
    return (int)result;
}
```


[Java]leetcode6 ZigZag Conversion

The string `"PAYPALISHIRING"` is written in a zigzag pattern on a given number of rows like this: (you may want to display this pattern in a fixed font for better legibility)

```
P   A   H   N
A P L S I I G
Y   I   R
```

And then read line by line: `"PAHNAPLSIIGYIR"`

Write the code that will take a string and make this conversion given a number of rows:

```
string convert(string text, int nRows);
```

convert("PAYPALISHIRING", 3) should return "PAHNAPLSIIGYIR".

题意是：给你一个字符串，他是以Z形式表示出来的字符串顺序，然后你以行的形式读出来。给定了字符串的行数。

解题思路：我们发现第一行和最后一行都很好处理，就是每次读取j+2*nRows-2，就是中间的行则需要单独处理。

正如以上的例子，我们发现，我们可以每次选区PAYP作为一个单元处理。就是选取nRows+nRows-2=2nRows-2个长度处理。

那么中间元素的位置关系是j+2*nRows-2i-2(i表示第几行)

ok,代码如下：

```
public String convert(String s, int nRows)
{
 if (s == null || nRows == 1)
  return s;
 int len = s.length();
 if (len <= nRows)
  return s;
 StringBuffer res = new StringBuffer();
 int size = 2 * nRows - 2;//每次处理的长度
 for (int i = 0; i < nRows; i++) {//每一行的元素
  char ch;
  for (int j = i; j < len; j += size) {
   ch = s.charAt(j);
   res.append(ch);
   if (i != 0 && i != nRows - 1) {//如果是中间元素单独处理
    int tmp = j + size - 2 * i;
    if (tmp < len) {
     ch = s.charAt(tmp);
     res.append(ch);
    }
   }
  }
 }
  return res.toString();
}
```

[Java]leetcode173 Binary Search Tree Iterator

Implement an iterator over a binary search tree (BST). Your iterator will be initialized with the root node of a BST.

Calling next() will return the next smallest number in the BST.

Note: next() and hasNext() should run in average O(1) time and uses O(h) memory, where h is the height of the tree.

题意：写一个二叉查找树的迭代器，实现hasNext()和next()的功能。next()每次返回二叉树中未访问的最小值。要求的平均时间复杂度是O(1)和空间复杂度是O(h)。

解题思路：next()每次返回二叉树中未访问的最小值。也就是将二叉树中序遍历，并将对应值输出。这里面注意的是空间和时间的复杂度。

```
public class BSTIterator {//将中序遍历的功能嵌查在整个程序中
  TreeNode current;
  Stack<TreeNode> stack;
  public BSTIterator(TreeNode root) {
      current=root;
      stack=new Stack<TreeNode>();
      while(current!=null)//因为只可能是左节点才是最小值，将所有左子树节点入栈，保证空间复杂度是O(h)
      {
          stack.push(current);
          current=current.left;
      }
  }
  /** @return whether we have a next smallest number */
  public boolean hasNext() {
   return !stack.isEmpty();
  }
  /** @return the next smallest number */
  public int next() {
    current=stack.pop();
    int res=current.val;//这一步已经得到最小值
    current=current.right;//但要考虑到右子树的遍历
    while(current!=null)
     {
       stack.push(current);
       current=current.left;
     }
     return res;
    }
}
```


[JAVA]LeetCode199 Binary Tree Right

Side View

Given a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

For example:
Given the following binary tree,
1 <—
/ \
2 3 <—
\ \
5 4 <—
You should return [1, 3, 4].
题目的含义是：从右边看，我们能看到的节点有哪些，相当于几何中投影的概念。
解题思路：1）深度遍历，先记录右边节点的最大高度。只有大于最大高度的节点才能打印出来。（我们自己思维去判断的时候，应该更倾向这种逻辑）
2）层次遍历，将最右边的节点存入list表中。这种想法写程序更容易理解。
实现1）思路代码如下所示：

```
public List<Integer> rightSideView(TreeNode root) {
    Stack<TreeNode> stack=new Stack<TreeNode>();
    Stack<TreeNode> stack1=new Stack<TreeNode>();//存储所有的节点，用来计算节点所在的高度
    List<Integer> list=new ArrayList<Integer>();
    TreeNode current=root;
    int curHeight=1;
    int rightHeight=0;
    while(current!=null||!stack.isEmpty())
    {
       if(current!=null)
       {
       if(curHeight>rightHeight)
       {
          list.add(current.val);
       }
       stack.push(current);
       stack1.push(current);
       current=current.right;
       ++curHeight;
       }else
       {
         current=stack.pop();
         //计算当前current节点所在的height值
         while(!stack1.isEmpty())
         {
            if(stack1.peek()==current)
            {
               curHeight=stack1.size();
      if(curHeight>rightHeight)rightHeight=curHeight;
               break;
            }
            stack1.pop();
```

```
            }
            current=current.left;
            ++curHeight;
          }
      }
      return list;
  }
```

实现2）思路程序如下：

```
public List<Integer> rightSideView(TreeNode root) {
    List<Integer> list=new ArrayList<Integer>();
    if(root==null)return list;
    Queue<TreeNode> queue=new LinkedList<TreeNode>();
    TreeNode current=root;
    queue.offer(current);
    queue.offer(null);//相当于用null来做每层节点的间隔符
    while(!queue.isEmpty())
    {
      current=queue.poll();
      if(current!=null)
      {
         if(queue.peek()==null)
         {
             list.add(current.val);
         }
         if(current.left!=null)queue.add(current.left);
        if(current.right!=null)queue.add(current.right);
      }else
      {
         if(queue.isEmpty())
         {
             break;
         }else
         {
             queue.add(null);
         }
      }
    }
    return list;
  }
```


[Java]LeetCode96 Unique Binary Search Trees

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example,
Given n = 3, there are a total of 5 unique BST's.

给定一个n值，那么从1.2.3....N共N个数，能构建几棵二叉排序树。

解题思路就是：遍历1至N个数，求它们每个元素作为根节点的情况，根据中序遍历的思想，小于根节点的排在左边，大于根节点的排在右边。求出左边子二叉树的构建数目和右二叉树的构建数目，相乘就得到该二叉树的数目了。该题是不是感觉又可以根据递归的思路来解了。

递归思路：

```
public int numTrees(int n) {
    if(n==0||n==1)return 1;
    if(n==2)return 2;
    int num=0;
    for(int i=1;i<=n;i++)//根节点
    {
       num+=numTrees(i-1)*numTrees(n-i);//左子树二叉排序树的构建数目*右子树二叉排序树的构建数目
    }
    return num;
  }
```

虽然逻辑是对的，但是leetcode提示timeLimited。也就是递归超时了。

那我们试一试将已经作为根节点求解的构建数目先用数组保存起来，然后后面的节点求解的时候再调用。

1）先定义一个array

2）array[0]表示节点数为0的情况，array[0]=1；

3）array[1]表示1个节点的求解只有一种情况：左节点的个数和右节点个数都为0即a[0]*a[0];

4）array[2]表示2个节点的求解只有两种情况：左节点的个数和右节点个数都为a[1]*a[0]，a[0]*a[1]

5）array[3]表示3个节点的求解只有三种情况：左节点的个数和右节点个数为a[2]*a[0]，a[1]*a[1]，a[0]*a[2]

以此类推

我们用代码实现如下

```
public int numTrees(int n) {
      int[] array=new int[n+1];
      array[0]=1;
      for(int i=1;i<=n;i++)
      {
        for(int j=0;j<i;j++)
        {
         array[i]+=array[j]*array[i-j-1];
        }
      }
      return array[n];
  }
```


LeetCode125 Sum Root to Leaf Numbers

Sum Root to Leaf Numbers

Given a binary tree containing digits from `0-9` only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path `1->2->3` which represents the number `123`.

解题思想：二叉树遍历，判断如果是叶子节点，就把路径的节点转换为整数。这里面我用了二个栈

stack用于返回上个节点

stack1用于记录每次根节点到叶子的所有节点，stack1中节点和stack一样添加节点，但是弹出的时机不一样。要注重分析。

```
public int sumNumbers(TreeNode root) {
    if(root==null)return 0;
    Stack<TreeNode> stack=new Stack<TreeNode>();
    Stack<TreeNode> stack1=new Stack<TreeNode>();
    StringBuilder builder=new StringBuilder();
    int sum=0;
    TreeNode current=root;
    while(current!=null||!stack.isEmpty())
    {
        if(current!=null)
        {
            stack.push(current);
            stack1.push(current);
            current=current.left;
        }else{
            current=stack.pop();
            if(current.right==null&&t.left==null)
            {
                for(int i=0;i<stack1.size();i++)
                {
                builder.append(stack1.get(i).val);
                }
                sum+=Integer.parseInt(builder.toString());
                builder.delete(0, builder.length());
                stack1.pop();//叶子节点计算完弹出</span><span style="color:#333333;">
            }else{
               while(!stack1.isEmpty())
                {
                 if(stack1.peek()==current)break;//当current存在右节点时stack中的current的节点已经弹出
                 stack1.pop();                  //但是stack1中的current节点不能弹出，我们要保持路径上的完整节点
                }
            }
            current=current.right;
        }
    }
    return sum;
  }
```


[JAVA]LeetCode106 Construct Binary Tree from Inorder and Postorder Traversal

根据中序和后序遍历结果构建二叉树

解题思路：递归调用，但需要注意的是先根据根节点建右树再建左树。

步骤：1）后序的最后一个节点肯定是树的根节点，取值建根节点,postIndex=postorder.length-1;

2）在中序中找该根节点值的位置，为inIndex

3)根据inIndex将中序分为左子树节点和右子树节点。

4）发现postIndex的前一位是树的右子树的节点。所以在建树的时候，我们要先建右树再建左树，postIndex--;

5)重复2）

```
public class Solution {
    Map<Integer,Integer> inMap=new TreeMap<Integer,Integer>();
    int postIndex=0;
   public TreeNode buildTree(int[] inorder, int[] postorder) {
      if(inorder.length==0||postorder.length==0)return null;
      postIndex=postorder.length-1;
      for(int i=0;i<inorder.length;i++)
      inMap.put(inorder[i],i);
      return createTree(postorder,0,inorder.length-1);
   }
   TreeNode createTree(int[] postorder,int inStart,int inEnd)
   {
       if(inStart>inEnd ||postIndex<0)return null;
       TreeNode root=new TreeNode(postorder[postIndex--]);
       if(inStart==inEnd)return root;
       int inIndex=inMap.get(root.val);
       root.right=createTree(postorder,inIndex+1,inEnd);
       root.left=createTree(postorder,inStart,inIndex-1);
       return root;
   }
}
```

[JAVA]LeetCode105 Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

根据前序遍历和中序遍历，构建二叉树。假设树中不存在相同的节点

解题思路：根据前序遍历的元素，每次都可以将中序遍历划分为左子树节点和右子树节点。考虑用递归的方法解题。

```
public class Solution
{
 Map<Integer,Integer> inMap=new TreeMap<Integer,Integer>();//将中序的值放入map中，可以很容易找
 int preIndex=0;//记录每棵树的下标
public TreeNode buildTree(int[] preorder, int[] inorder)
{
 if(preorder.length==0||inorder.length==0)return null;
 for(int i=0;i<inorder.length;i++)
  {
    inMap.put(inorder[i],i);
  }
    return createTree(preorder,0,inorder.length-1);
}
 public TreeNode createTree(int[] preorder,int inStart,int inEnd)
 {
    if(inStart>inEnd)return null;
    TreeNode root=new TreeNode(preorder[preIndex++]);
    if(inStart==inEnd)return root;
    int inIndex=inMap.get(root.val);
    root.left=createTree(preorder,inStart,inIndex-1);
    root.right=createTree(preorder,inIndex+1,inEnd);
    return root;
 }
}
```

LeetCode114 Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

将二叉树变成扁平的，类似于LinkedList的只有右子树的线性链表

基本思路:这不是前序遍历么，将节点存入list中，然后重新构建一个右子树。这就简单了。

```
public void flatten(TreeNode root) {
    Stack<TreeNode> stack=new Stack<TreeNode>();
    List<TreeNode> list=new LinkedList<TreeNode>();
    TreeNode current=root;
    int i=0;
    while(!stack.isEmpty()||current!=null)//前序遍历节点存入list中
    {
        if(current!=null)
        {
            stack.push(current);
            list.add(current);
            current=current.left;
        }else
        {
            current=stack.pop();
            current=current.right;
        }
    }
    current=root;//将list中的节点重新构建右子树，注意之前的链接还在，所以构建的时候，要将左孩子清空
    i=1;
    while(i<list.size())
    {
        current.left=null;
        current.right=list.get(i);
        current=current.right;
        i++;
    }
}
```

LeetCode94 Binary Tree Inorder Traversal[Java]

非递归的二叉树中序遍历

使用stack来解这道题

步骤：1、初始化空栈stack

2、初始化current指针，先指向root

3、将current压入栈，并使得current=current.left直到current为空

4、如果current为空，stack不为空

1）将值从stack中弹出，current=stack.pop();

2)list.add(current);

3)current=current.right

4)重复3

5、如果current为空，stack也为空，遍历结束

按照以上步骤，实现代码如下：

```
public List<Integer> inorderTraversal(TreeNode root)
{
 List<Integer> list = new ArrayList<Integer>();
 Stack<TreeNode> stack = new Stack<TreeNode>();
 TreeNode current = root;
 while(!stack.isEmpty()||current!=null)
 {
  if(current!=null)
```

```
  {
   stack.push(current);
   current=current.left;
  }else
  {
   current=stack.pop();
   list.add(current.val);
   current=current.right;
  }
 }
 return list;
}
```


LeetCode 147 Insertion Sort List

问题：Sort a linked list using insertion sort.链表的插入。

这题的考察点：1)插入排序(以某个数为基准，一般是第一个数，将后面的数与它比较，放在前边还是后边。。。。)

2)其次是链表的插入，需要单向的遍历

```
public ListNode insertionSortList(ListNode head) {
        if(head==null||head.next==null)return head;//head为空或只有一个数返回head
        int minNum=Integer.MIN_VALUE;//新建一个结点作为头结点，头结点的值为整数的最小值
        ListNode p=new ListNode(minNum);
        p.next=head;
        head=p;
        p=p.next;
        ListNode q=p.next;//从第二数开始与前边的数比较
        ListNode r,rN;//用于遍历前部分排序好的链表，确定q指针所指节点插入的位置，每次都从头结点开始
        while(q!=null)
        {
         r=head;
         rN=r.next;
         if(q.val<p.val)
         {
          while(rN!=p && rN.val<=q.val)//rN所指值小于q,r和rN都后移
          {
            r=rN;
            rN=rN.next;

         }//跳出循环表示找到了q值插入的位置，并修改相应结点的链接
         r.next=q;
```

```
      q=q.next;
      r=r.next;
      r.next=rN;
      p.next=q;
        }else//说明q指针的值小于p,不用修改链接,指针后移
        {
          p=q;
          q=q.next;
        }
       }
     return head.next;  //去掉新建的节点，返回head.next
   }
```


LeetCode146 LRU Cache

Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.
get(key) - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
set(key, value) - Set or insert the value if the key is not already present. When the cache reached its capacity, it should invalidate the least recently used item before inserting a new item.
该题就是设计一个LRU(最近最少访问的cache)Cache,支持get和set方法。当缓冲区满的时候，根据最近最少访问的原则将新的item插入即可。 思路：键值对的插入很容易想到Map，LinkedListMap就有两种插入方式 accessOrder布尔值为true时为 access-order（访问顺序）进行插入，也就是按照最近最少访问的原则插入，false时为insertion-order（插入顺序），即按插入的先后顺序进行插入。
A special {@link #LinkedHashMap(int,float,boolean) constructor} is provided to create a linked hash map whose order of iteration is the order in which its entries were last accessed, from least-recently accessed to most-recently (access-order). This kind of map is well-suited to building LRU caches.
通过阅读源代码，我们发现LinkedHashMap(int,float,boolean) 构造方法是解决该题的关键。

```
public LinkedHashMap(int initialCapacity, float loadFactor,boolean accessOrder)
{
      super(initialCapacity, loadFactor);
      this.accessOrder = accessOrder;
}
```

构造函数中的三个参数分别是，initialCapacity初始容量，loadFactor荷载因子，accessOreder是否为访问顺序

当我们选择accessOrder方式时，当Cache达到最大容量时要删除最近最少访问的元素，所以要重载该方法。
The {@link #removeEldestEntry(Map.Entry)} method may be overridden to impose a policy for removing stale mappings automatically when new mappings are added to the map.
removeEldestEntry当新的映射添加到map中，我们可能需要重载该方法来自动删除旧的映射。这里的映射就是键值对的意思。
怎样重载，在注释中也给出了例子。让我们来看看注释。
Returns true if this map should remove its eldest entry.This method is invoked by put and putAll after inserting a new entry into the map. It provides the implementor with the opportunity to remove the eldest entry each time a new one is added. This is useful if the map represents a cache: it allows the map to reduce memory consumption by deleting stale entries.
如果map需要删除旧的entry,return返回true。当用put和putAll方法初入新的entrys时该方法就会被调用。如果该map代表cache时。通过删除旧的entry来达到节省内存消耗的作用。
Sample use: this override will allow the map to grow up to 100 entries and then delete the eldest entry each time a new entry is added, maintaining a steady state of 100 entries.
例如，当map增加到100entries后，需要删除旧的entry来添加新的entry，维持100固定的长度。

```
private static final int MAX_ENTRIES = 100;
protected boolean removeEldestEntry(Map.Entry eldest)
{
    return size() > MAX_ENTRIES;
}
```

好了，结合少量的源码分析和题目的理解，我们来实现LRUCache的功能如下。

```
import java.util.LinkedHashMap;
import java.util.Map;

public class LRUCache<K, V> extends LinkedHashMap<Integer, Integer>
{
  private static final long serialVersionUID = 1L;
  private int capacity;
  public LRUCache(int capacity)
  {
     super(capacity, 0.75f, true);
     this.capacity = capacity;
  }
  @Override
  protected boolean removeEldestEntry(Map.Entry<Integer, Integer> eldest)
  {
     // TODO Auto-generated method stub
     return size() > capacity;
  }
  public int get(int key)
  {
     // TODO Auto-generated method stub
```

```
        if(super.get(key)==null)return -1;
        return super.get(key);
    }
    public void  set(int key, int value)
    {
        // TODO Auto-generated method stub
        super.put(key,value);
    }
}
```


LeetCode61Rotate List

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`.

解题思路：

1、先遍历整个链表求出总长度length

2、根据k求出链表的新的head在什么位置，length-k的位置

3、假如p指向length-k-1的地方，head=p.next,然后将p.next=null,因为它要作为链表的结尾

3、然后从新的head节点遍历到末尾，链接到初始head节点的位置。

出错的地方：题目中k的值可以大于链表的长度。那要怎么解决呢？实际上当k值大于链表长度时，旋转k次的结果等于k%length后的结果。我们在这稍加判断即可。

```
if (head == null)
  return null;
 int length=1;
 int i = 1;
 ListNode p = head;
 ListNode q = head;
 while(p.next!=null)
 {
  p=p.next;
  length++;
```

```
}
p=head;
if(n>length)n=(n%length);
int k=length-n;
if(k==0)return head;
while (p.next != null)
{
 if (i > k-1)
  break;
 p = p.next;
 i++;
}
if (p.next != null)
{
 head = p.next;
 p.next = null;
}
else
 return head;
p = head;
while (p.next != null)
 p = p.next;
p.next = q;
return head;
```


Remove Duplicates from Sorted List II

将有序链表中的重复元素全部删除，比如1->2->2->3结果是1->3，1->1->1结果是空。

1、我们先新建ListNode节点

```
public class ListNode
{
int val;
ListNode next;

ListNode(int x)
{
 val = x;
 next = null;
}
}
```

2、为了方便测试，我们再新建List表，当输入-100时，退出循环。

```
public class List
{
ListNode head,tail;
int length;
List()
{
 head=null;
 tail=null;
}
public boolean isEmpty()//判断链表是否为真
{
 return head==null;
}

public void addHead(int val)
{
 head=new ListNode(val);
 head.next=null;
 if(tail==null)tail=head;
}

public void addToTail(int val)
{
 if(!isEmpty())
 {
  tail.next=new ListNode(val);
  tail=tail.next;
 }
 else
 {
  head=tail=new ListNode(val);
 }
}

public void print(ListNode head)
{
 ListNode p=head;
 while(p!=null)
 {
  System.out.print(p.val+" ");
  p=p.next;
 }
}

public ListNode createList()
{
 Scanner sc=new Scanner(System.in);
 int number;
 while(sc.hasNext())
 {
  number=sc.nextInt();
  if(number==-100)break;
  else
```

```
  {
   addToTail(number);
  }
 }
 return head;
}
}
```

3、接下来就是分析算法

这题的难点在1、如果重复元素出现在开头，我们就需要修改头指针。

2、如果中间重现不同的重复元素，如1->2->2->3->3->3->4->4，也就是假如在1的地方有一个指针，那该指针在遇到什么情况下才移动呢。

3、该题会设计到多个指针的使用，会有结尾元素单独处理的情况。

笔者解决该题思路是：1、我们首先解决头指针需要修改的情况，也就是head.val==head.next.val

2、head指针不需要动了之后，p=head，q=head.next，r=q.next，那接下来，我们就需要判断p在什么情况下移动就可以了。

如下图分析：

不管在什么情况下q、r指针都要移动。p移动的情况如下：

1、p.val!=q.val&&q.val!=r.val(p.next=q)(1->2->3->3的情况)

2、如上图q.val!=r.val&&r.val!=r.next.val(p.next!=q)

好，分析到这，接下来我们可以写程序了。

程序如下：

```
/**
* Definition for singly-linked list.
* public class ListNode {
*     int val;
*     ListNode next;
*     ListNode(int x) {
*         val = x;
*         next = null;
*     }
* }
*/
public class Solution {
  public ListNode deleteDuplicates(ListNode head)
{
if (head == null || head.next == null)
  return head;
 ListNode p = head;
 ListNode q = head.next;

 if (head.val == head.next.val)
 {
  while (q.next != null)
  {
   if (p.val != q.val && q.val != q.next.val)
   {
    head = q;
    p = q;
    q = q.next;
    break;
   }
   p = q;
   q = q.next;
  }
 }
 if (p.val == q.val)
  return null;
 else
 {
  if (q.next == null)
  {
   if (head.next != q)
    return q;
   else
   {
    return head;
   }
  }

 }
              //上面代码是修改头指针
              ListNode r = q.next;
 while (r.next != null)
 {
  if (p.val != q.val && q.val != r.val)
  {
    if (p.next == q)
```

```
      p = q;
                            else{
                                if (r.val != r.next.val)
                                      p.next=r;
                                }
  }
  q = r;
  r = r.next;
 }
 if (q.val == r.val)
  p.next = null;
 else
 {
  if (p.next != q)
   p.next = r;
 }
 return head;
}
}
```