**单链表的部分反转**

问题：

         给出一个单链表（不带头节点）和一个数K，请翻转此单链表？

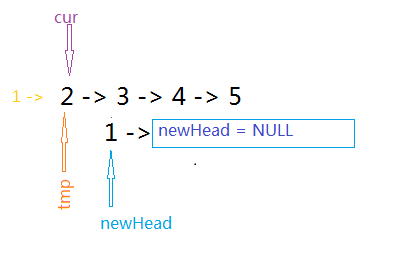
例如：1->2->3->4->5       k = 0;   翻转过后：1->2->3->4->5

          1->2->3->4->5       K = 2;   翻转过后：2->1->4->3->5

          1->2->3->4->5       K = 10; 翻转过后：5->4->3->2->1

        在讨论链表的‘部分’翻转问题之前，先回顾一下链表的逆置问题，如何将一个链表进行逆置？我们以前采用的方式是‘摘节点头插’的方法。假设右链表1->2->3->4->5，新建一个节点指针newHead=NULL，用cur指针指向链表的头节点，将摘下的节点用tmp进行保存，然后将tmp和newHead进行连接，将newHead指向tmp的位置上，如此循环直到cur为NULL结束。

■下面是‘摘节点头插’的简单图示：

[](http://s4.51cto.com/wyfs02/M01/80/85/wKiom1dDu22j--UMAAAOqK4yEFE776.png" \t "http://blog.csdn.net/ttyue_123/article/details/_blank)

1. //链表节点的结构
2. **struct** Node
3. {
4. **int** data;
5. Node\* \_next;
6. };
8. **void** Reverse(Node\* list)
9. {
10. **if** (list == NULL)
11. {
12. **return**;
13. }
14. Node\* cur = list;
15. Node\* newHead = NULL;
16. **while** (cur)
17. {
18. Node\* tmp = cur;
19. cur = cur->\_next;
20. tmp->\_next = newHead;
21. newHead = tmp;
22. }
23. }

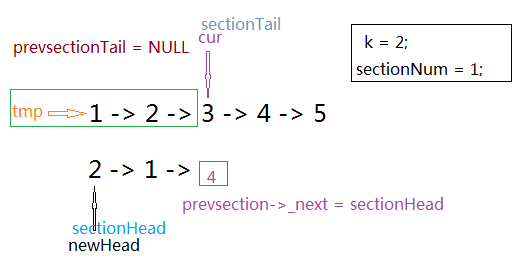
通过上面的解释，读者应该对链表的逆置应该不会陌生了，下面就还看一下链表的‘部分’翻转的问题：

        链表的‘部分’翻转，它和链表的逆置还是不一样的，对于给定的K，其实就是没此逆置K个节点，然后连接上后面翻转过后的K个节点，若最后剩余的不足K个节点，同样也对其进行翻转，然后进行连接。所以我们可以借助一下‘摘节点头插’的方式，分部分进行‘摘节点头插’。

        假设1->2->3->4->5,k = 2; 即就是将1和2进行翻转，3和4进行翻转，5进行翻转，然后将1和2翻转后的结果与3和4翻转后的结果和5翻转后的结果进行连接，得到2->1->4->3->5.

        在完成代码之前，**先考虑链表若为空的情况，针对这个问题，如果k =0/1时的情况，k若大于链表节点的总个数又是什么情况？**当链表为空时，可以直接返回，或者k=0/1时，就不需要对其进行翻转，也可以直接进行返回。若K大于链表节点的总个数，即就是相当于对链表进行逆置。

        在完成链表‘部分’的翻转，需要sectionHead（指向‘部分’的头节点）、sectionTail（指向‘部分’的尾节点）、cur（指向整个链表）、newHead（指向翻转完成的链表头节点）、prevsectionTail（指向这一部分之前部分的尾节点）、tmp（用来做中间保存节点的指针）这几个节点指针，sectionNum用来标识翻转的是第几部分。下面是‘部分’翻转的简单图示：

[](http://s4.51cto.com/wyfs02/M00/80/87/wKiom1dDxTWTny68AAAksTjP87k777.png" \t "http://blog.csdn.net/ttyue_123/article/details/_blank)

■下面是‘部分’翻转的实现代码：

1. Node\* RolloverList(Node\* list, **int** k)
2. {
3. **if** (k <= 1 || list == NULL)
4. {
5. **return** list;
6. }
8. Node\* cur = list;      //指向链表的头节点
9. Node\* newHead = NULL;      //指向逆置后的链表
10. Node\* sectionHead = NULL;     //指向需要逆置部分的头节点
11. Node\* sectionTail = NULL;      //指向需要逆置部分的尾节点
12. Node\* prevsectionTail = NULL;    //指向前部分尾节点的指针
13. **int** sectionNum = 0;   //用来标记链表翻转到第几部分
15. **while** (cur)
16. {
17. **int** count = k;     //记录部分逆置的节点数
18. prevsectionTail = sectionTail;
19. sectionTail = cur;
21. **while** (count-- && cur != NULL)     //使用的方法还是摘节点头插(进行部分逆置)
22. {
23. Node\* tmp = cur;
24. cur = cur->\_next;
25. tmp->\_next = sectionHead;     //sectionHead相当于链表逆置中的newHead（先将部分进行逆置）
26. sectionHead = tmp;
27. }
29. ++sectionNum;   //统计逆置了几部分
30. //如果sectionNum为1时，逆置的部分为第一部分，就应该确定逆置后的链表的头节点
31. **if** (sectionNum == 1)
32. {
33. newHead = sectionHead;
34. }
35. **else**    //证明逆置的为链表后面的部分，需要将后面的部分和前面的部分进行连接起来
36. {
37. prevsectionTail->\_next = sectionHead;
38. }
39. }
41. //出循环cur == NULL
42. sectionTail->\_next = NULL;
43. **return** newHead;
44. }

题目：反转链表从m-n位置的结点

For example:  
Given1->2->3->4->5->NULL, m = 2 and n = 4,

return1->4->3->2->5->NULL.

从第二到第四的结点被反转了。

其中m和n满足条件：  
1 ≤ m ≤ n ≤ length of list.

对于链表的问题，根据以往的经验一般都是要建一个dummy node，连上原链表的头结点，这样的话就算头结点变动了，我们还可以通过dummy->next来获得新链表的头结点。这道题的要求是只通过一次遍历完成，就拿题目中的例子来说，变换的是2,3,4这三个点，那么我们可以先取出2，用front指针指向2，然后当取出3的时候，我们把3加到2的前面，把front指针前移到3，依次类推，到4后停止，这样我们得到一个新链表4->3->2, front指针指向4。对于原链表连说，有两个点的位置很重要，需要用指针记录下来，分别是1和5，因为当2,3,4被取走时，原链表就变成了1->5->NULL，要把新链表插入的时候需要这两个点的位置。1的位置很好找，因为知道m的值，我们用pre指针记录1的位置，5的位置最后才能记录，当4结点被取走后，5的位置需要记下来，这样我们就可以把倒置后的那一小段链表加入到原链表中。代码如下：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode(int x) : val(x), next(NULL) {}

\* };

\*/

class Solution {

public:

ListNode \*reverseBetween(ListNode \*head, int m, int n) {

ListNode \*dummy = new ListNode(-1);

dummy->next = head;

ListNode \*cur = dummy;

ListNode \*pre, \*front, \*last;

for (int i = 1; i <= m - 1; ++i) cur = cur->next;

pre = cur;

last = cur->next;

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

cur = pre->next;

pre->next = cur->next;

cur->next = front;

front = cur;

}

cur = pre->next;

pre->next = front;

last->next = cur;

return dummy->next;

}

};

仍是逆序，仍考虑到用辅助空间stack.

将m-n的结点依次入栈，并标记与入栈结点相邻的前后两个结点pfirst和psecond

(若m==1，pfirst=null，不管n是否为length of list，对psecond的情况无影响。)

代码如下：

1. **import** java.util.\*;
2. **public** **class** Solution {
3. **public** ListNode reverseBetween(ListNode head, **int** m, **int** n) {
4. **if**(head==**null**)
5. **return** **null**;
6. **if**(m==n)
7. **return** head;
8. Stack<ListNode> stack=**new** Stack();
9. //将m-n的结点入栈，将前后相邻的两个结点标记;
10. **int** num=1;
11. ListNode pfirst=**null**;
12. ListNode psecond=**null**;
13. ListNode p=head;
14. //特殊情况，m==1时，头结点变更;
15. **if**(m==1)
16. pfirst=**null**;
17. **for**(;num<=n;num++)
18. {
19. //记录pfirst;
20. **if**(num<m)
21. {
22. **if**(num==m-1)
23. {
24. pfirst=p;
25. }
26. p=p.next;
27. }
28. **else** **if**(num>=m&&num<=n)
29. {
30. stack.push(p);
31. p=p.next;
32. }
33. }
34. //记录psecond,psecond的一般情况仍适用于n=length of list的特殊情况;
35. psecond=p;
36. //开始操作链表;
37. **if**(pfirst==**null**)
38. {
39. head=stack.pop();
40. pfirst=head;
41. }
42. **while**(!stack.empty())
43. {
44. pfirst.next=stack.pop();
45. pfirst=pfirst.next;
46. }
47. pfirst.next=psecond;
48. **return** head;
49. }
50. }

public class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode(int x) {

val = x;

}

}

算法实现类

import java.util.Deque;import java.util.LinkedList;import java.util.List;

\*/public class Solution {

public List<List<Integer>> zigzagLevelOrder(TreeNode root) {

List<List<Integer>> result = new LinkedList<>();

if (root == null) {

return result;

}

// 遍历标志，0表示从左到右，1表示从右到左

int flag = 0;

TreeNode node;

// 记录每一层的元素

List<Integer> lay = new LinkedList<>();

// 双向队列，当作栈来使用，记录当前层待处理结点

Deque<TreeNode> stack = new LinkedList<>();

// 记录下一层待处理结点

Deque<TreeNode> nextStack = new LinkedList<>();

stack.add(root);

while (!stack.isEmpty()) {

// 删除栈顶元素

node = stack.removeLast();

// 结果入队

lay.add(node.val);

// 如果当前是从左到右遍历，按左子树右子树的顺序添加

if (flag == 0) {

if (node.left != null) {

nextStack.addLast(node.left);

}

if (node.right != null) {

nextStack.addLast(node.right);

}

}

// 如果当前是从右到左遍历，按右子树左子树的顺序添加

else {

if (node.right != null) {

nextStack.addLast(node.right);

}

if (node.left != null) {

nextStack.addLast(node.left);

}

}

// 当前层已经处理完了

if (stack.isEmpty()) {

Deque<TreeNode> temp = nextStack;

nextStack = stack;

stack = temp;

// 标记下一层处理的方向

flag = 1 - flag;

// 保存本层结果

result.add(lay);

// 创建新的链表处理下一层的结果

lay = new LinkedList<>();

}

}

return result;

}

}