第一课 链表

林沐

内容概述

1.8道经典链表常考题目

例1-a:链表逆序(easy)

例1-b:链表逆序2(medium)

例2:链表求交点 (easy)

例3:链表求环(medium)

例4:链表划分(medium)

例5:复杂链表的复制(hard)

例6-a:2个排序链表归并(easy)

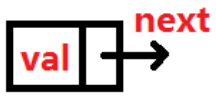
例6-b:K个排序链表归并(hard)

2.详细讲解题目多种解题方法、代码实现

3.一些学习与找工作的建议

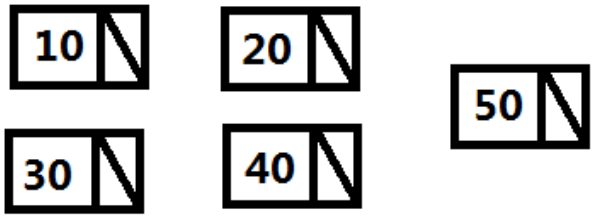
预备知识:链表基础

```
#include <stdio.h>

struct ListNode {
    int val;   //存储元素的数据域
    ListNode *next;
};      //存储下一个节点地址的指针域

int main(){
    ListNode a;
    ListNode b;
    ListNode c;
    ListNode d;
    ListNode e;
    a.val = 10;
    b.val = 20;
    c.val = 30;
    d.val = 40;
    e.val = 50;
    a.next = &b;
        1
    c.next = &d;
    d.next = &e;
        2
    ListNode *head = &a;
    while(head){
        printf("%d\n", head->val);
            3
    }
    return 0;
}
```

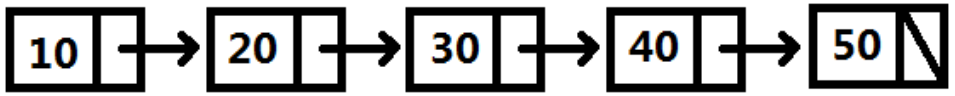


1分钟时间填写代码！

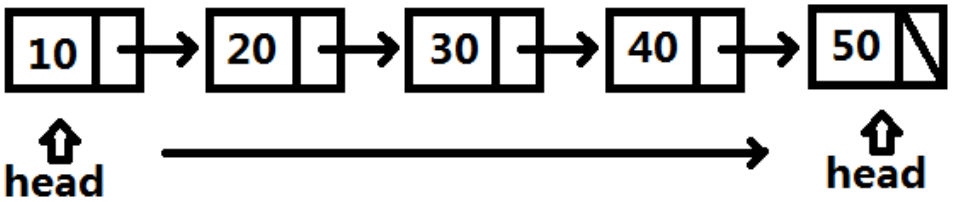
1.创建5个节点:



2.将它们连接在一起:

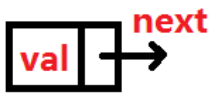


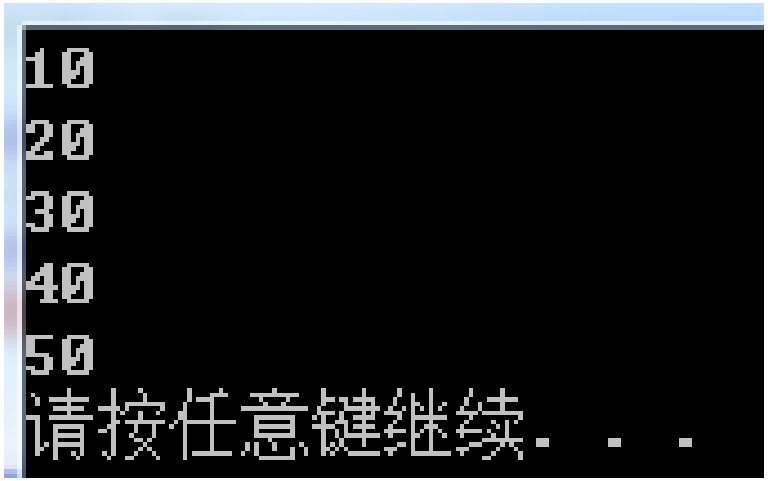
3.遍历它们，并打印节点的值:



预备知识:链表基础

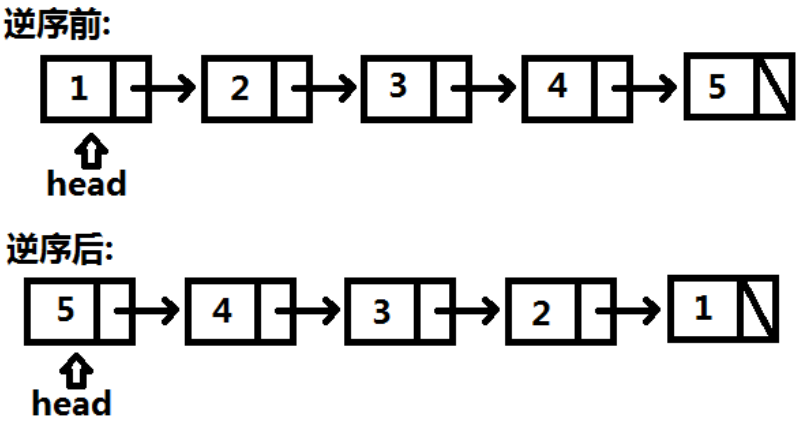
```
#include <stdio.h>

struct ListNode {
    int val;   //存储元素的数据域
    ListNode *next;
};      //存储下一个节点地址的指针域

int main(){
    ListNode a;
    ListNode b;
    ListNode c;
    ListNode d;
    ListNode e;
    a.val = 10;
    b.val = 20;
    c.val = 30;
    d.val = 40;
    e.val = 50;
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    e.next = NULL;
    ListNode *head = &a;
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    return 0;
}
```

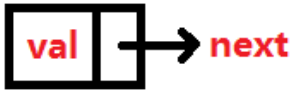




例1-a:链表逆序-a

已知链表**头节点**指针head，将**链表**逆序。(不可申请**额外**空间)






```
struct ListNode {
    int val; //数据域
    ListNode *next; //指针域
    ListNode(int x) : val(x), next(NULL) {}
};          //构造函数

class Solution {
public:                          //链表头节点指针
    ListNode* reverseList(ListNode* head) {
    }
                //返回链表逆序后的头节点指针
};
```

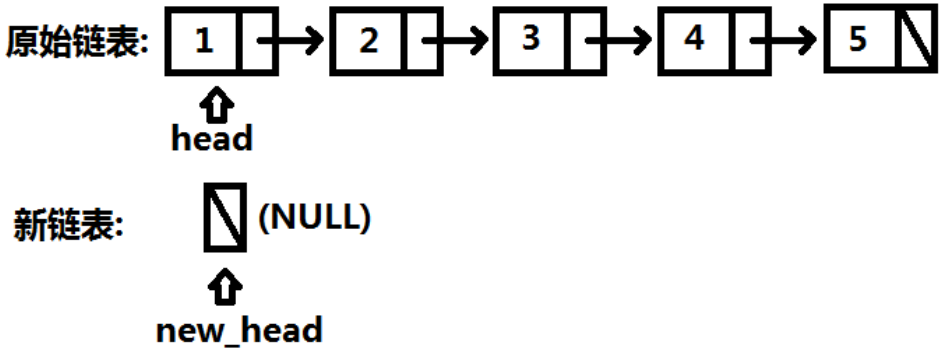
选自 **LeetCode 206. Reverse Linked List**

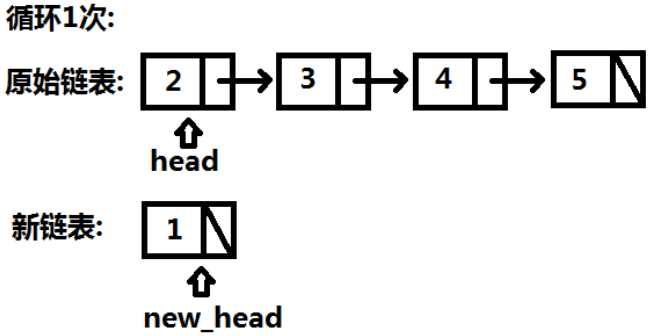
https://leetcode.com/problems/reverse-linked-list/description/

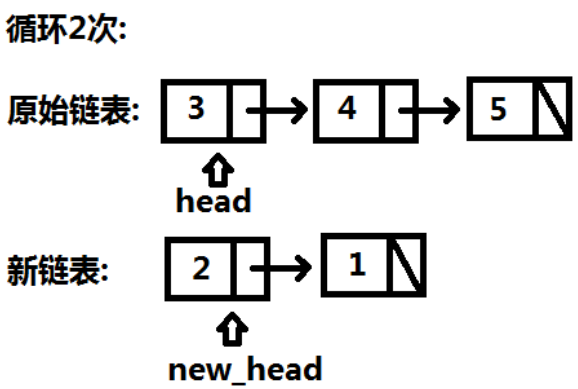
难度:**Easy**

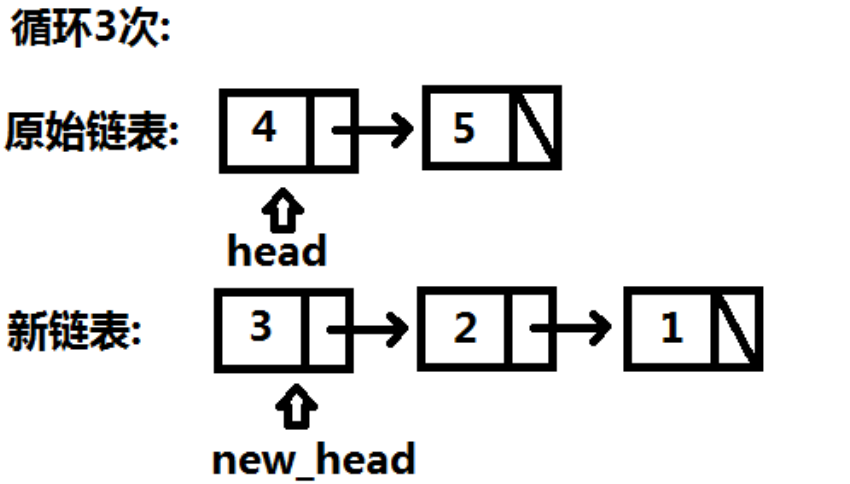
例1-a:思路

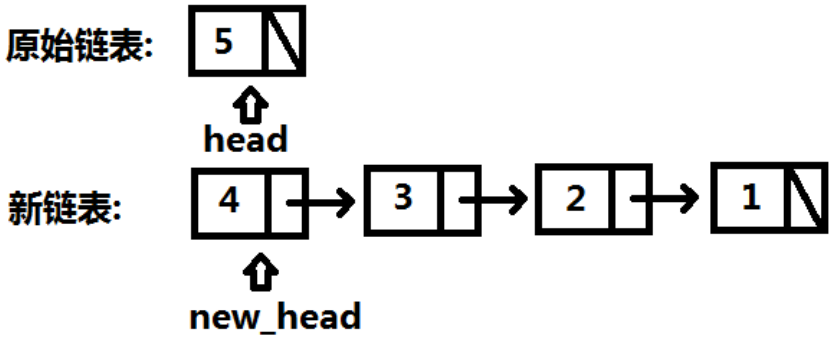
依次遍历链表节点，每遍历一个节点即逆置一个节点

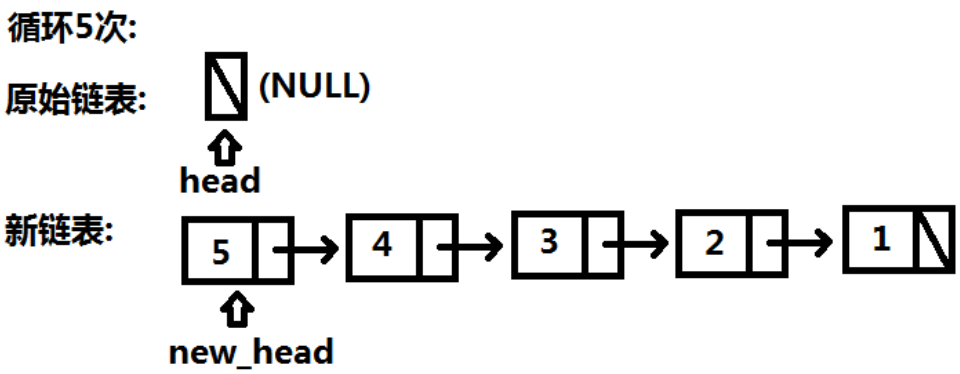




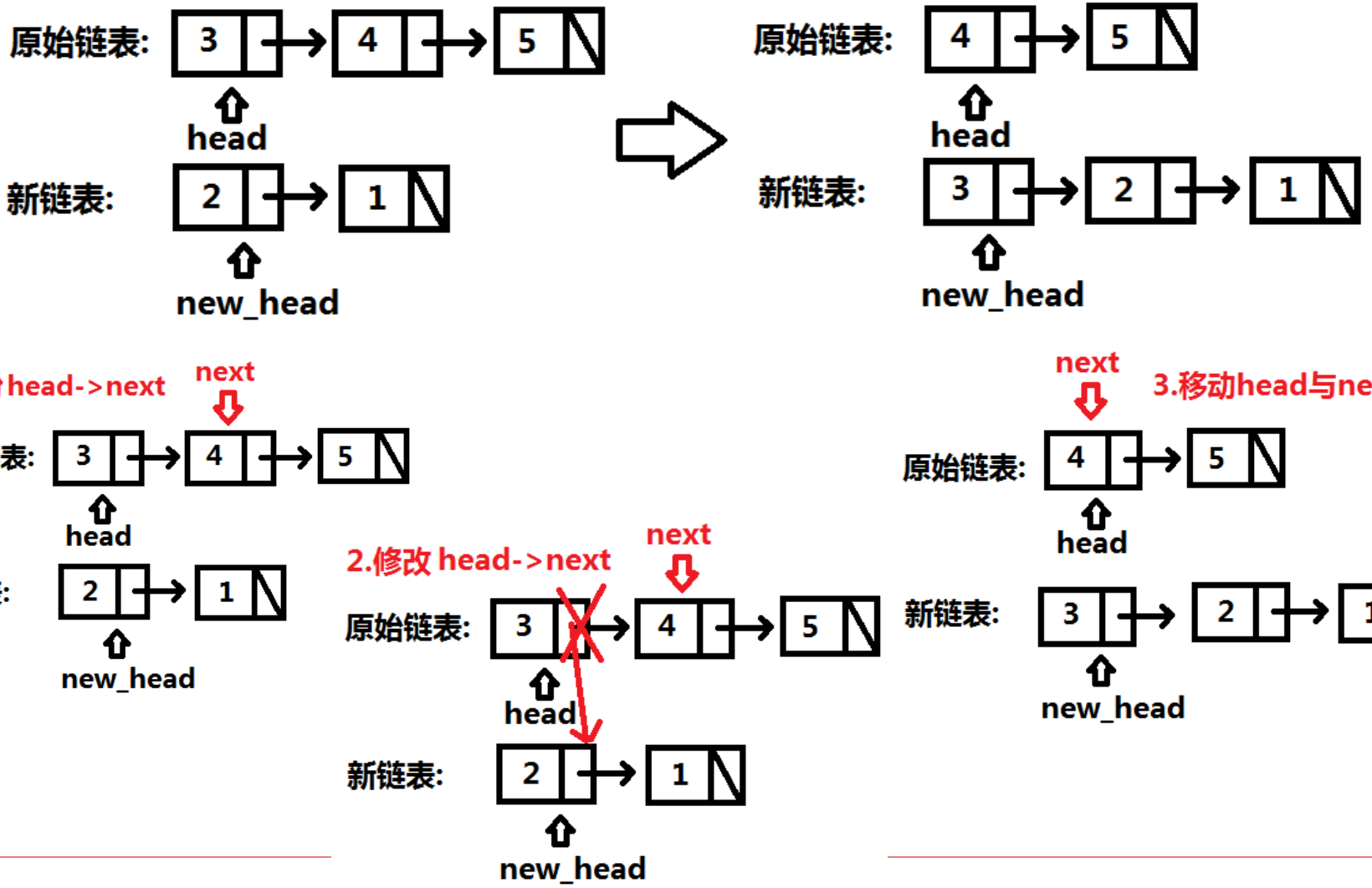








例1-a:思路


原始链表:
3
4
5
head
新链表:
2
1
new_head
原始链表:
4
5
head
新链表:
3
2
1
new_head
1.备份head->next
next
原始链表:
3
4
5
head
新链表:
2
1
new_head
2.修改 head->next
next
原始链表:
3
4
5
head
新链表:
2
1
new_head
next
3.移动head与new_head
原始链表:
4
5
head
新链表:
3
2
1
new_head

例1-a:课堂练习

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *new_head = NULL; //指向新链表头节点的指针
        while(head){
            1
            2
            3
            head = next;  //遍历链表
        }
        return new_head; //返回新链表头节点
    }
};
```

3分钟时间填写代码，**有问题随时提出！**

例1-a:实现

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *new_head = NULL; //指向新链表头节点的指针
        while(head){
            ListNode *next = head->next;   //备份head->next
            head->next = new_head;         //更新head->next
            new_head = head;               //移动new_head
            head = next;                   //遍历链表
        }
        return new_head;                   //返回新链表头节点
    }
};
```

例1-a:测试与leetcode提交结果

```
int main(){
    ListNode a(1);
    ListNode b(2);
    ListNode c(3);
    ListNode d(4);
    ListNode e(5);
    a.next = &b;
    b.next = &c; //将节点简单的链接，进行测试
    c.next = &d; //无需构造复杂的链表操作(插入,删除)
    d.next = &e;
    Solution solve;
    ListNode *head = &a;
    printf("Before reverse:\n");
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    head = solve.reverseList(&a);
    printf("After reverse:\n");
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    return 0;
}
```

```
Before reverse:
1
2
3
4
5
After reverse:
5
4
3
2
1
请按任意键继续. . .
```

Reverse Linked List

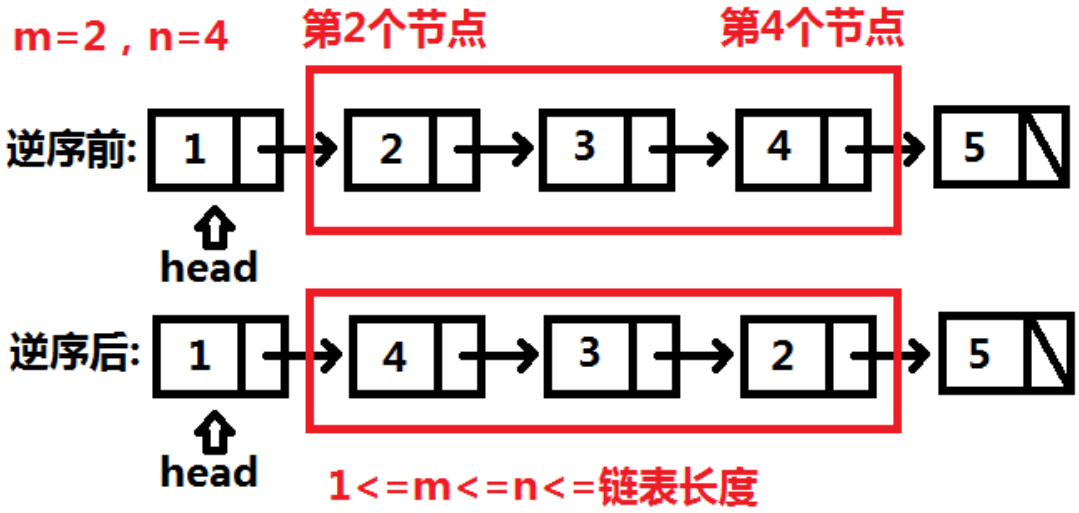
Submission Details

27 / 27 test cases passed. Status: Accepted

Runtime: 6 ms Submitted: 0 minutes ago

例1-b:链表逆序-b

已知链表**头节点**指针head，将**链表**从位置m到n逆序。(不申请**额外**空间)



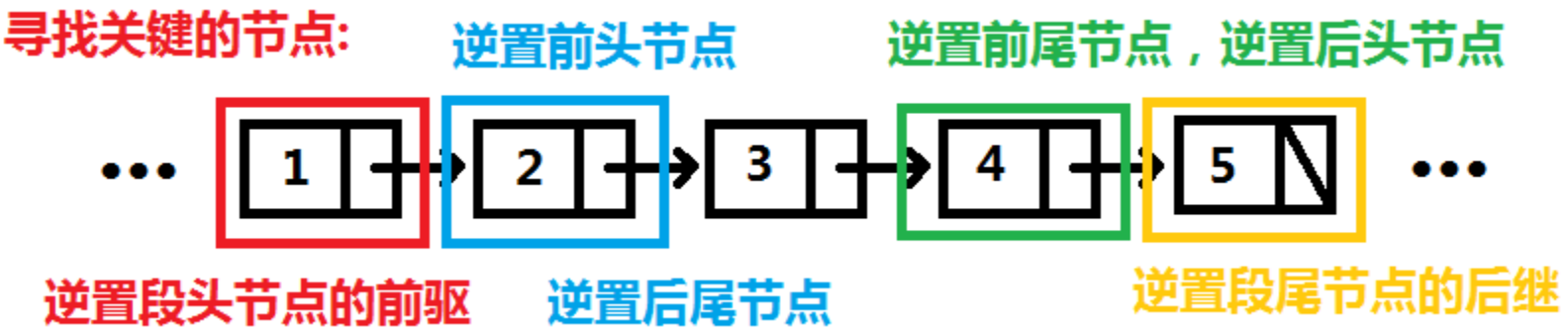

```
class Solution {
public:                                  //链表头指针          //从m逆至到n
    ListNode* reverseBetween(ListNode* head, int m, int n) {
    }
};
```

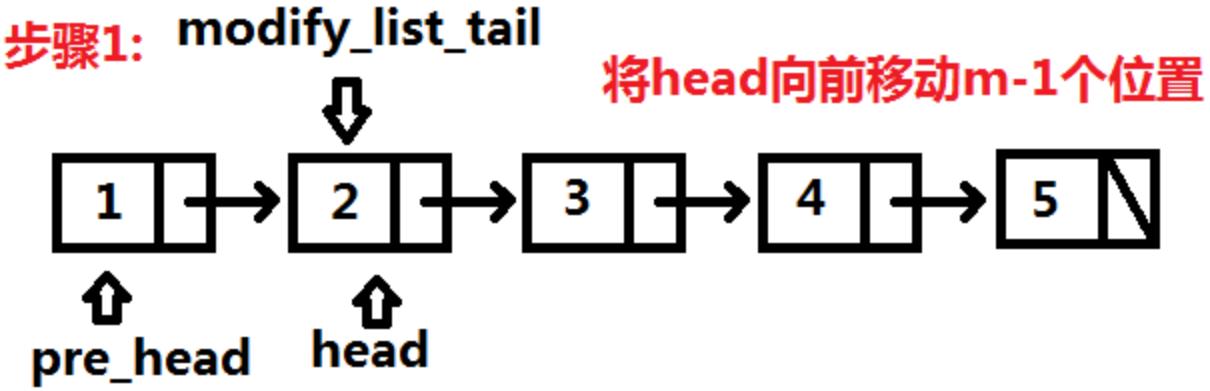
选自 **LeetCode 92. Reverse Linked List II**

https://leetcode.com/problems/reverse-linked-list-ii/description/

难度:**Medium**

例1-b:思路

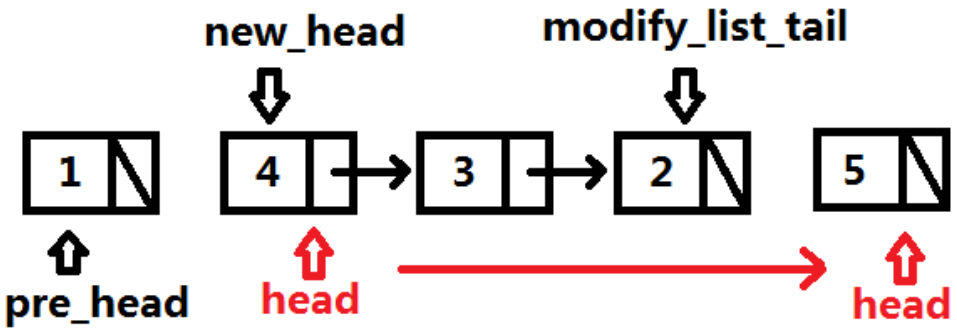

寻找关键的节点:
逆置前头节点
逆置前尾节点，逆置后头节点
1
2
3
4
5
逆置段头节点的前驱
逆置后尾节点
逆置段尾节点的后继


步骤1:
modify_list_tail
将head向前移动m-1个位置
1
2
3
4
5
pre_head
head
找到开始逆置的节点，记录该节点前驱、该节点

例1-b:思路

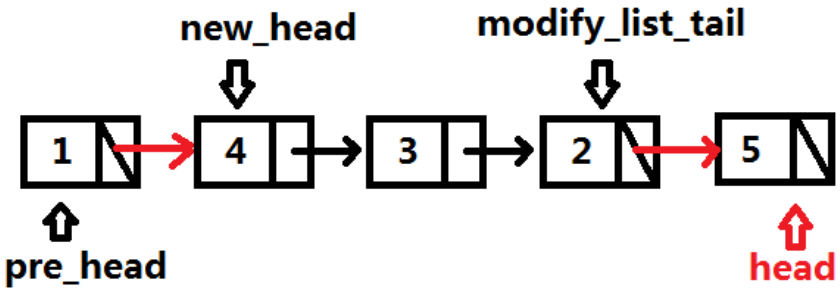
步骤2:

从head开始，逆置change_len = n-m+1个节点



步骤3:

将pre_head与new_head连接，modify_list_tail与head连接



思考:

1.最终结果应该**返回**哪个节点?

2.如果m=1时，有什么**特殊**的?

例1-b:课堂练习

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int m, int n) {
        int change_len = n - m + 1; //计算需要逆置的节点个数
        ListNode *pre_head = NULL; //初始化开始逆置的节点的前驱
        ListNode *result = head; //最终转换后的链表头节点，非特殊情况即为head
        while(head && --m){                    //将head向前移动m-1个位置
            1
            head = head->next;
        }                                      //将modify_list_tail指向当前的head，即逆置后的链表尾
        ListNode *modify_list_tail = head;
        ListNode *new_head = NULL;
        while(head && change_len){             //逆置change_len个节点
            ListNode *next = head->next;
            head->next = new_head;
            new_head = head;
            head = next;
            2
        }
        3
        if (pre_head){                         //如果pre_head不空，说明不是从第一
            4                                  //个节点开始逆置的m>1
        }
        else{
            5
        }
        return result;
    }
};
```

5分钟时间填写代码，

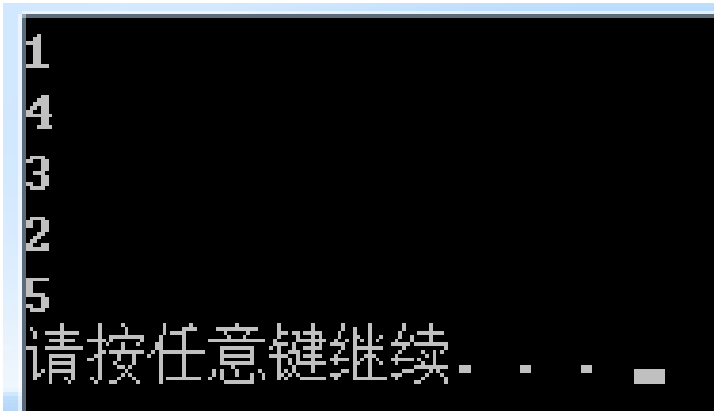
有问题随时提出！

例1-b:实现

```
class Solution {
public:
    ListNode* reverseBetween(ListNode* head, int m, int n) {
        int change_len = n - m + 1; //计算需要逆置的节点个数
        ListNode *pre_head = NULL; //初始化开始逆置的节点的前驱
        ListNode *result = head; //最终转换后的链表头节点，非特殊情况即为head
        while(head && --m){
            pre_head = head;
            head = head->next;
        }
        //将head向前移动m-1个位置
        //记录head的前驱
        //将modify_list_tail指向当前的head，即逆置后的链表尾
        ListNode *modify_list_tail = head;
        ListNode *new_head = NULL;
        while(head && change_len){
            ListNode *next = head->next;
            head->next = new_head;
            new_head = head;
            head = next;
            change_len--;
        }
        //逆置change_len个节点
        //每完成一个节点逆置，change_len--;
        modify_list_tail->next = head;
        //连接逆置后的链表尾与逆置段的后一个节点
        if (pre_head){
            pre_head->next = new_head;
        }
        else{
            result = new_head;
        }
        //如果pre_head不空，说明不是从第一个节点开始逆置的m>1
        //将逆置 链表开始的节点前驱与逆置后的头节点链接
        //如果pre_head为空，说明m==1从第一个节点开始逆置结果即为逆置后的头节点
        return result;
    }
};
```

例1-b:测试与leetcode提交结果

```
int main(){
    ListNode a(1);
    ListNode b(2);
    ListNode c(3);
    ListNode d(4);
    ListNode e(5);
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    Solution solve;
    ListNode *head = solve.reverseBetween(&a, 2, 4);
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    return 0;
}
```



Reverse Linked List II

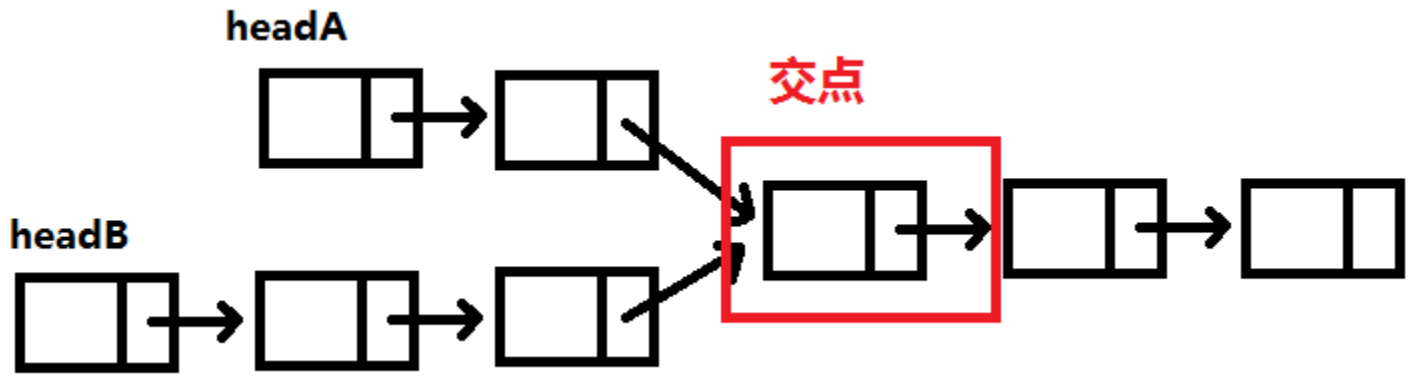
Submission Details

44 / 44 test cases passed. Status: Accepted

Runtime: **3 ms** Submitted: **0 minutes ago**

例2:求两个链表的交点

已知链表A的**头节点**指针headA，链表B的**头节点**指针headB，两个链表**相交**，求两链表**交点**对应的节点



选自 **LeetCode 160. Intersection of Two Linked Lists**

https://leetcode.com/problems/intersection-of-two-linked-lists/description/

难度:**Easy**

例2:题目要求

```
struct ListNode {
    int val; //数据域
    ListNode *next; //指针域
    ListNode(int x) : val(x), next(NULL){
    }           //构造函数
};
```


```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB){
    }           求两个链表的交点
};
```

链表A头节点指针　　链表B头节点指针

1.如果两个链表**没有交点**，则**返回NULL**

2.在求交点的过程中，**不可以破坏**链表的**结构**或者**修改**链表的**数据域**

3.可以确保传入的链表A与链表B**没有任何环**

4*.实现算法尽可能使**时间**复杂度**O(n)**，**空间**复杂度**O(1)**

例2:方法1的必备知识(set的使用)

```
#include <set>
int main(){
    std::set<int> test_set; //STL set
    const int A_LEN = 7;
    const int B_LEN = 8;
    int a[A_LEN] = {5, 1, 4, 8, 10, 1, 3};
    int b[B_LEN] = {2, 7, 6, 3, 1, 6, 0, 1};

    for (int i = 0; i < A_LEN; i++){
        test_set.insert(a[i]);
    }
    for (int i = 0; i < B_LEN; i++){
        if (test_set.find(b[i]) != test_set.end()){
            printf("b[%d] = %d in array A.\n", i, b[i]);
        }
    }
    return 0;
}
```

//STL set的使用

//O(nlogn)或O(n)的方法判断两数组是否有相同元素

//测试数组A与B的长度

//测试数组A与B，其中有重复元素

//将数组a的元素插入set

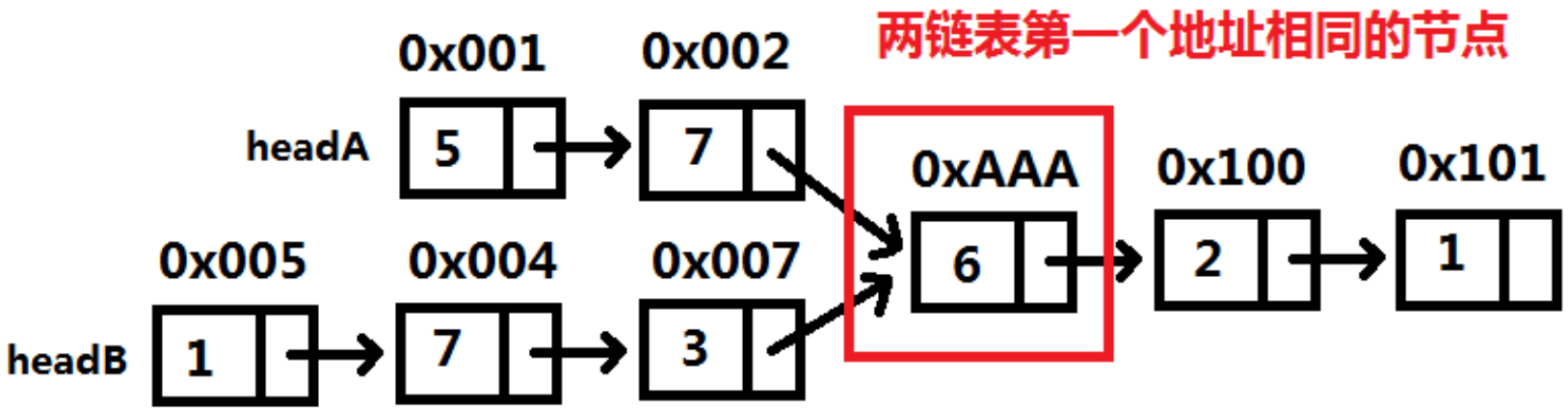
//检查数组b中的元素是否在set中

```
b[3] = 3 in array A.
b[4] = 1 in array A.
b[7] = 1 in array A.
请按任意键继续. . .
```

例2:思路1，使用set求交集

1.遍历链表A，将**A中**节点对应的**指针(地址)**，**插入set**

2.遍历链表B，将**B中**节点对应的**指针(地址)**，在set中**查找**，发现在set中的第一个节点地址，即是两个链表的**交点**。



例2:方法1，课堂练习

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        std::set<ListNode*> node_set;
        while(headA){
            1
            headA = headA->next;
        }
        while(headB){
            if (2){
                return headB;
            }
            3
        }
        return NULL;
    }
};
```

//设置查找集合node_set

//遍历链表A

//当在headB中找到第一个出现在node_set中的节点时

3分钟时间填写代码，**有问题随时提出！**

例2:方法1实现

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        std::set<ListNode*> node_set;
        while(headA){
            node_set.insert(headA);
            headA = headA->next;
        }
        while(headB){
            if (node_set.find(headB) != node_set.end()){
                return headB;
            }
            headB = headB->next;
        }
        return NULL;
    }
};
```

//设置查找集合node_set

//将链表A中的节点插入node_set

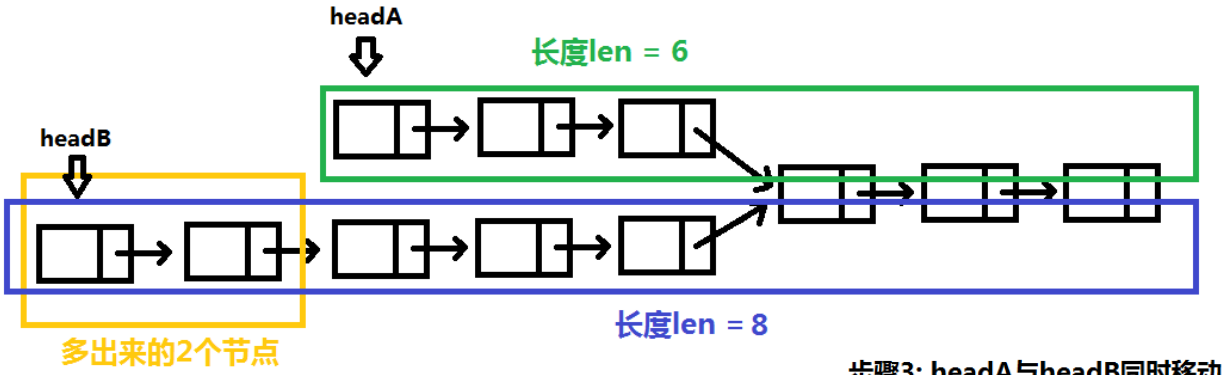
//遍历链表A

//当在headB中找到第一个出现在node_set中的节点时

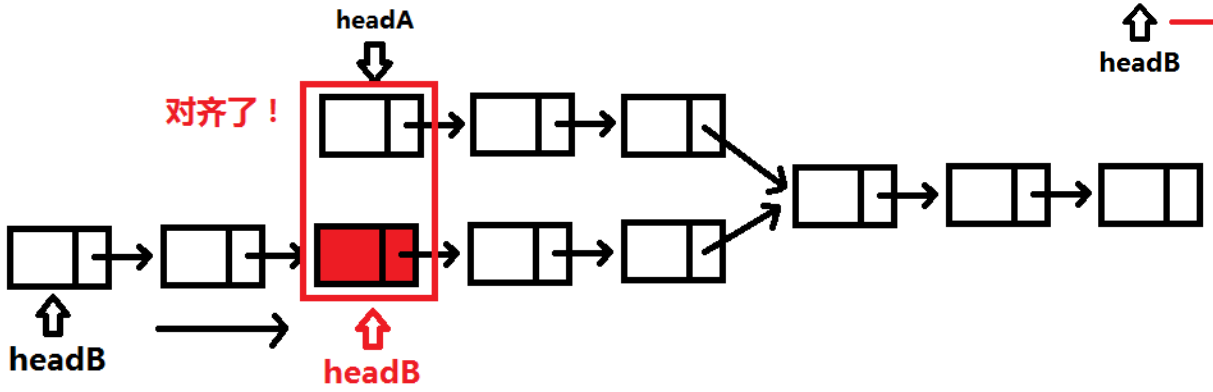
//遍历链表B

例2:思路2，空间复杂度O(1)

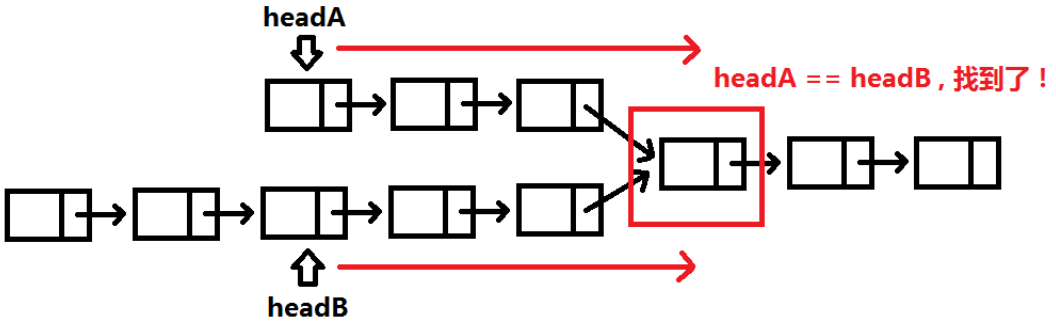
步骤1: 计算headA链表长度、计算headB链表长度，较长的链表多出的长度



步骤2: 将较长链表的指针移动到和较短链表指针对齐的位置



步骤3: headA与headB同时移动，当两指针指向同一个节点时，即找到了！



例2:方法2，课堂练习

```
int get_list_length(ListNode *head){
    int len = 0;
    while(head){ //遍历链表，计算链表长度
        1
        head = head->next;
    }
    return len;
}
```

```
ListNode *forward_long_list(int long_len,
                            int short_len, ListNode *head){
    int delta = long_len - short_len;
    while(head && delta){ //将指针向前移动至多出
        head = head->next;
        2                 节点个数后面的位置
    }
    return head;
}
```

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        int list_A_len = get_list_length(headA);
        int list_B_len = get_list_length(headB); //求A、B两个链表长度
        if (list_A_len > list_B_len){
            headA = forward_long_list(list_A_len, list_B_len, headA);
        }          //如果链表A长，移动headA到对应位置
        else{
            headB = forward_long_list(list_B_len, list_A_len, headB);
        }          //如果链表B长，移动headB到对应位置
        while(headA && headB){
            if ( 3 ){
                return headA;
            }
            headA = headA->next;
            headB = headB->next;
        }
        return NULL;
    }
};
```

3分钟时间填写代码，**有问题随时提出！**

例2:方法2，实现

```
int get_list_length(ListNode *head){
    int len = 0;
    while(head){
        len++;
        head = head->next;
    }
    return len;
}
```

//遍历链表，计算链表长度

```
ListNode *forward_long_list(int long_len,
                            int short_len, ListNode *head){
    int delta = long_len - short_len;
    while(head && delta){
        head = head->next;
        delta--;
    }
    return head;
}
```

//将指针向前移动至多出节点个数后面的位置

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        int list_A_len = get_list_length(headA);
        int list_B_len = get_list_length(headB);
        if (list_A_len > list_B_len){
            headA = forward_long_list(list_A_len, list_B_len, headA);
        }
        else{
            headB = forward_long_list(list_B_len, list_A_len, headB);
        }
        while(headA && headB){
            if (headA == headB){
                return headA;
            }
            headA = headA->next;
            headB = headB->next;
        }
        return NULL;
    }
};
```

//求A、B两个链表长度

//如果链表A长，移动headA到对应位置

//如果链表B长，移动headB到对应位置

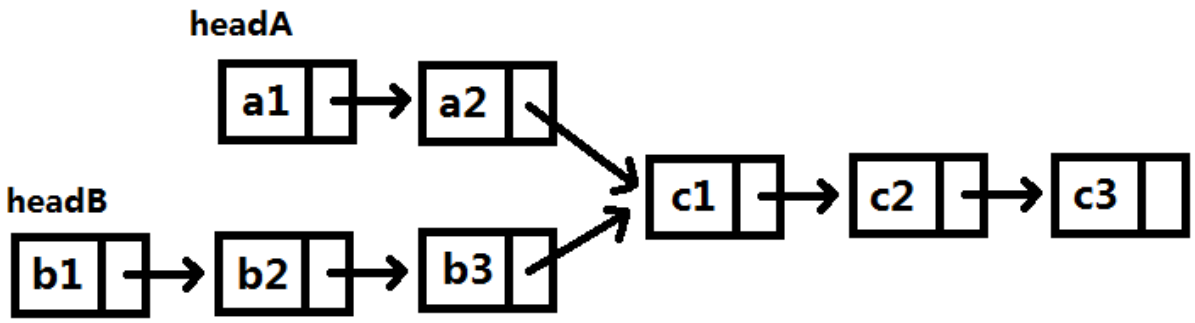
//当两指针指向了同一个节点时，说明找到了！

例2:测试与leetcode提交结果

```
int main(){
    ListNode a1(1);
    ListNode a2(2);
    ListNode b1(3);
    ListNode b2(4);
    ListNode b3(5);
    ListNode c1(6);
    ListNode c2(7);
    ListNode c3(8);
    a1.next = &a2;
    a2.next = &c1;
    c1.next = &c2;
    c2.next = &c3;
    b1.next = &b2;
    b2.next = &b3;
    b3.next = &c1;
    Solution solve;
    ListNode *result = solve.getIntersectionNode(&a1, &b1);
    printf("%d\n", result->val);
    return 0;
}
```

//将节点简单的链接一下，即可进行测试

//无需构造链表插入、删除等复杂方法




```
6
请按任意键继续. . .
```

Intersection of Two Linked Lists

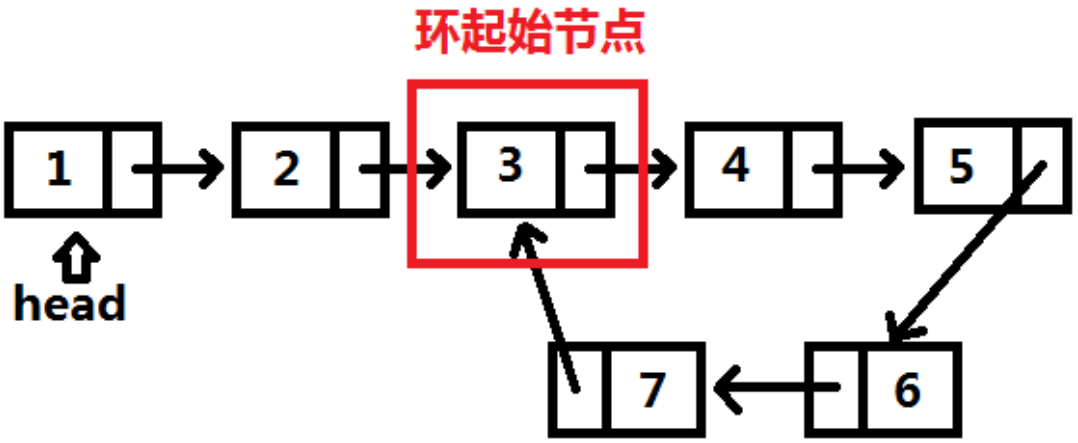
Submission Details

42 / 42 test cases passed. Status: Accepted

Runtime: 63 ms Submitted: 0 minutes ago

例3:链表求环

已知链表中**可能**存在**环**，若有环返回**环起始**节点，否则返回**NULL**。




```
class Solution {
public:
    ListNode *detectCycle(ListNode *head)
    }
};
```

选自 **LeetCode 141. Linked List Cycle　142.Linked List Cycle II**

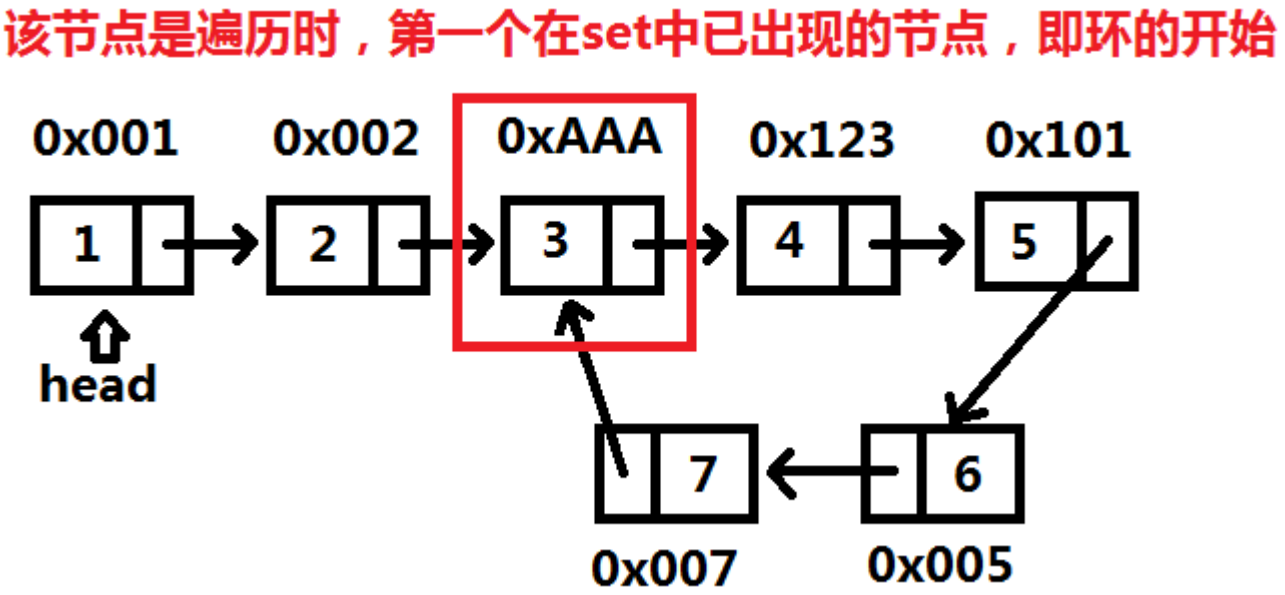
https://leetcode.com/problems/linked-list-cycle-ii/description/

难度:**Medium**

例3:思路1，使用set求环起始节点

1.遍历链表，将**链表中**节点对应的**指针(地址)，插入set**

2.在遍历时**插入节点前**，需要在set中**查找**，**第一个**在set中发现的节点地址，即是链表环的**起点**。



例3:方法1，课堂练习

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        std::set<ListNode *> node_set; //设置node_set
        while(head){   //遍历链表
            if (                1                ){
                return head;  //返回环的第一个节点
            }
            2
            head = head->next;
        }
        3
    }
};
```

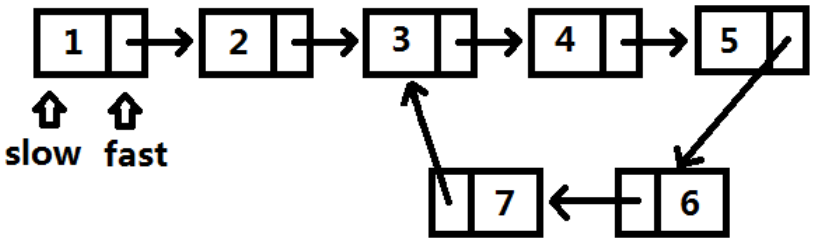
3分钟时间填写代码，**有问题随时提出！**

例3:方法1实现

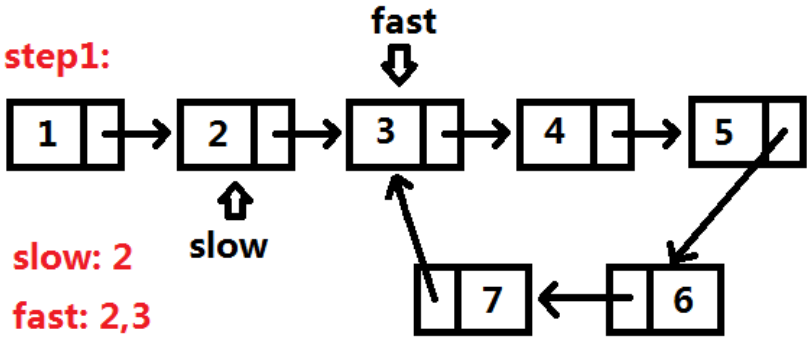
```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        std::set<ListNode *> node_set; //设置node_set
        while(head){   //遍历链表    //如果在node_set已经出现了
            if ( node_set.find(head) != node_set.end() ) {
                return head;  //返回环的第一个节点
            }
            node_set.insert(head);  //将节点插入node_set
            head = head->next;
        }
        return NULL;  //没有遇到环，则返回NULL
    }
};
```

例3:思路2，快慢指针赛跑

step0:



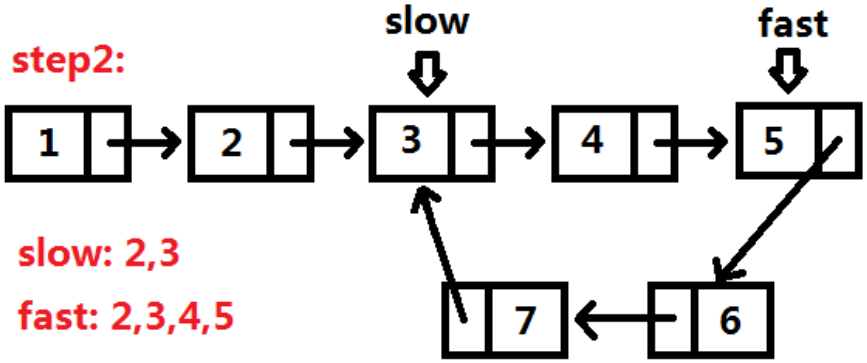
step1:



slow: 2

fast: 2,3

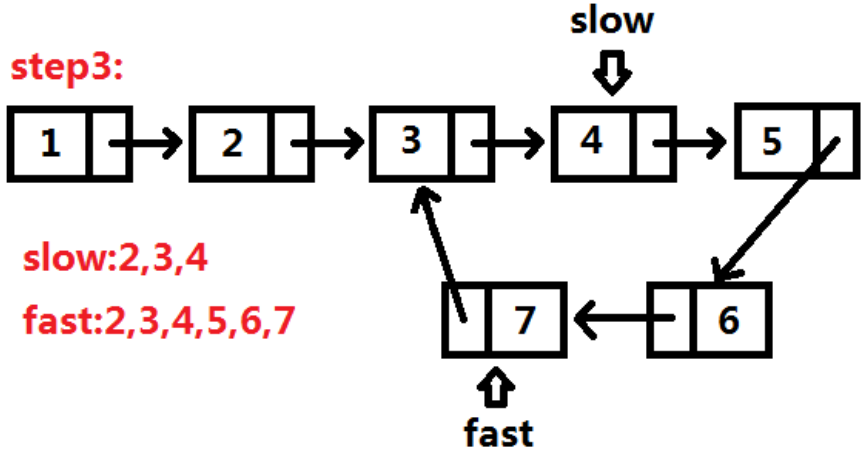
step2:



slow: 2,3

fast: 2,3,4,5

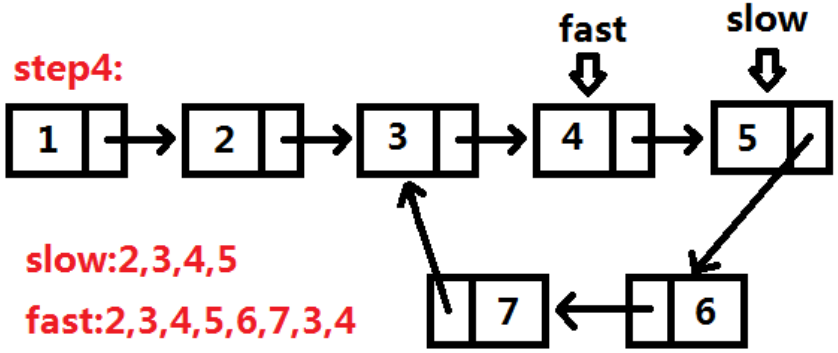
step3:



slow:2,3,4

fast:2,3,4,5,6,7

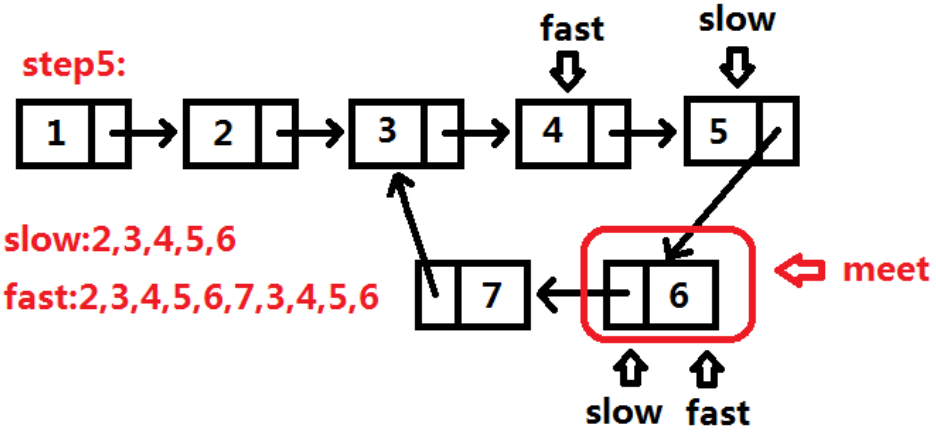
step4:



slow:2,3,4,5

fast:2,3,4,5,6,7,3,4

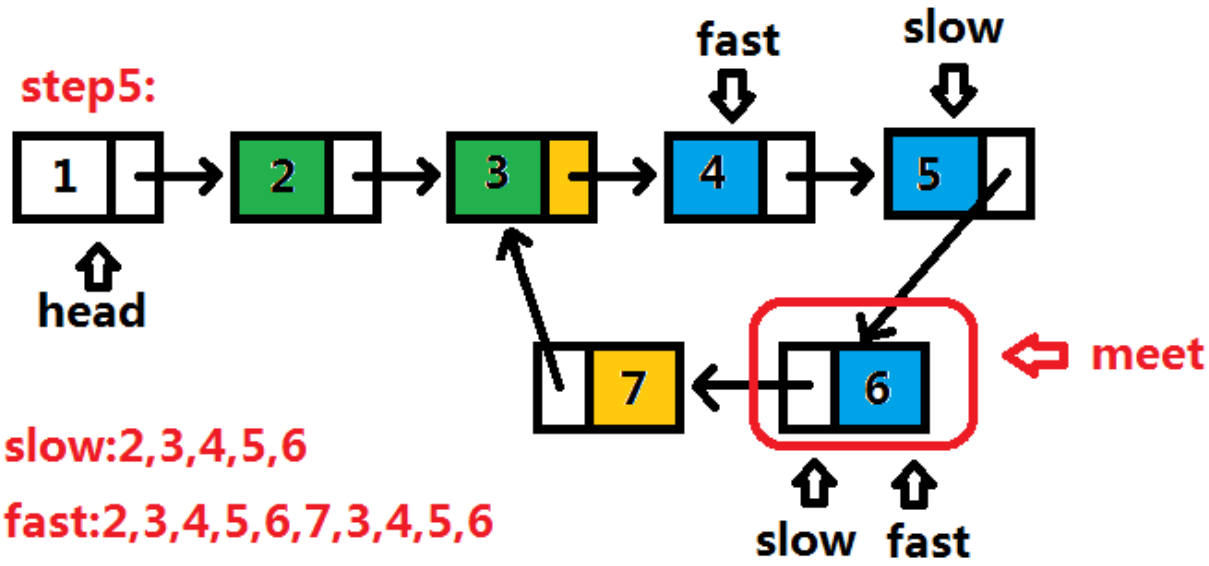
step5:



slow:2,3,4,5,6

fast:2,3,4,5,6,7,3,4,5,6

例3:思路2，快慢指针



设:

1)绿色 2,3 段 为 a

2)蓝色 4,5,6 段为b

3)橙色 7,3 段为c

slow = a + b

fast = a + b + c + b

由于 fast 走的路程 是slow的两倍

故:

2 *(a + b) = a + b + c +b

a = c

结论: 从head 与 meet 出发，两指针速度一样，相遇时即为环的起点

例3:方法2，课堂练习

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *fast = head; //快慢指针
        ListNode *slow = head;
        ListNode *meet = NULL; //相遇的节点
        while(fast){
            slow = slow->next;
            fast = fast->next; //slow与fast先各走一步
            if (!fast){
                    1
            }
                2
            if (fast == slow){
                    3
                break;
            }
        }
        if (meet == NULL){
                4
        }
        while(head && meet){
            if (     5     ){
                return head;
            }
            head = head->next; //head与meet每次走1步
            meet = meet->next;
        }
        return NULL;
    }
};
```

5分钟时间填写代码，**有问题随时提出！**

例3:方法2，实现

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *fast = head;  //快慢指针
        ListNode *slow = head;
        ListNode *meet = NULL;  //相遇的节点
        while(fast){
            slow = slow->next;  //slow与fast先各走一步
            fast = fast->next;
            if (!fast){
                return NULL;    //如果fast遇到链表尾，则返回NULL
            }
            fast = fast->next;  //fast再走1步
            if (fast == slow){
                meet = fast;    //fast与slow相遇，记录相遇位置
                break;
            }
        }
        if (meet == NULL){
            return NULL;        //如果没有相遇，则证明无环
        }
        while(head && meet){
            if (head == meet){  //当head与meet相遇，说明遇到环的起始位置
                return head;
            }
            head = head->next;  //head与meet每次走1步
            meet = meet->next;
        }
        return NULL;
    }
};
```

例3:测试与leetcode提交结果

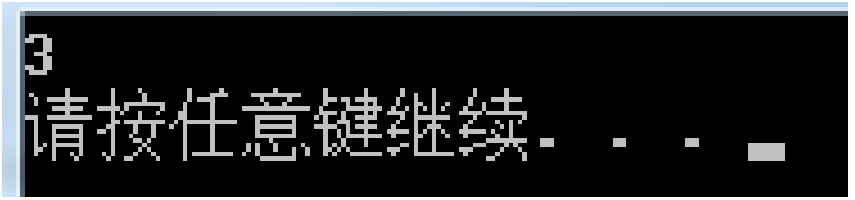
```
int main(){
    ListNode a(1);
    ListNode b(2);
    ListNode c(3);
    ListNode d(4);
    ListNode e(5);
    ListNode f(6);
    ListNode g(7);
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    e.next = &f;
    f.next = &g;
    g.next = &c;
    Solution solve;
    ListNode *node = solve.detectCycle(&a);
    if (node){
        printf("%d\n", node->val);
    }
    else{
        printf("NULL\n");
    }
    return 0;
}
```

Linked List Cycle II

Submission Details

16 / 16 test cases passed. Status: Accepted

Runtime: **9 ms** Submitted: **0 minutes ago**

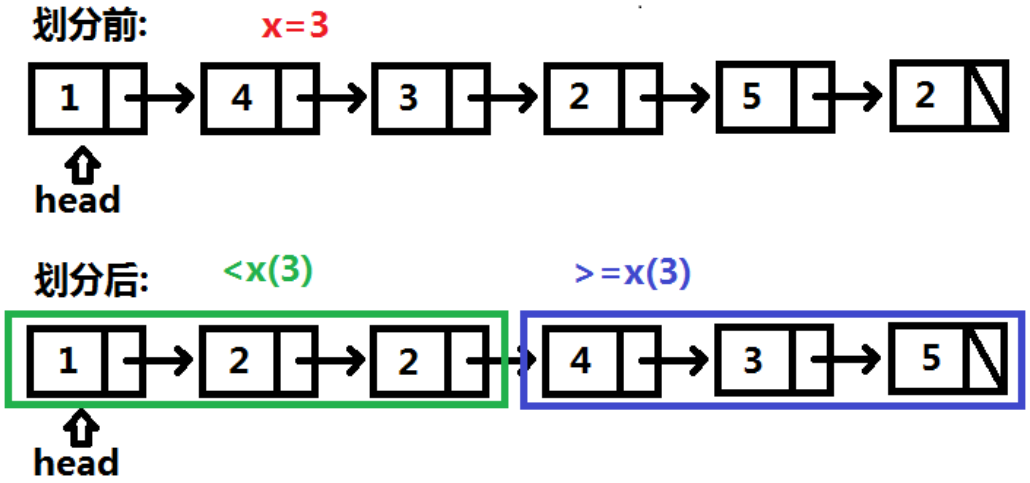


课间休息10分钟

课前小问题解答

例4:链表划分

已知链表头指针head与**数值x**，将所有**小于x**的节点放在**大于或等于x**的节点**前**，且**保持**这些节点的**原来**的相对位置。



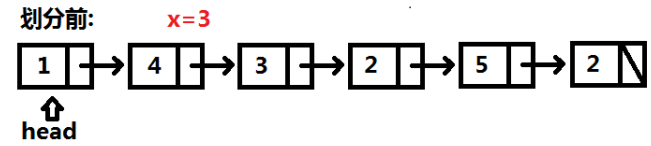

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
    }
};
```

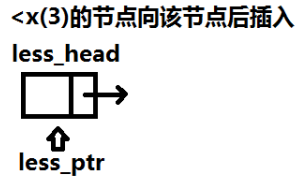
选自 **LeetCode 86. Partition List**

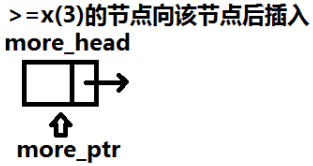
https://leetcode.com/problems/partition-list/description/

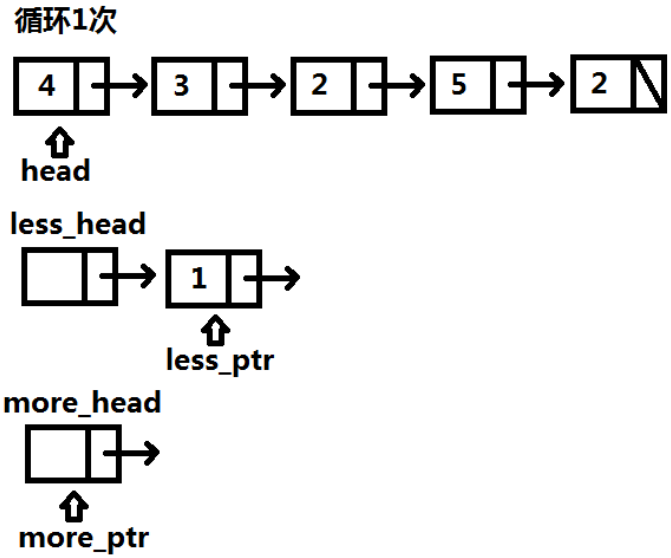
难度:**Medium**

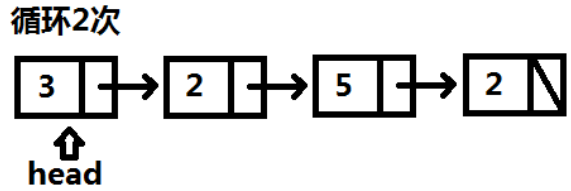
例4:思路，巧用临时头节点

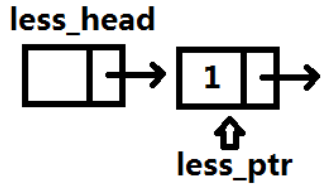


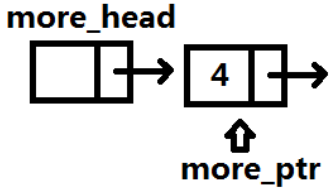


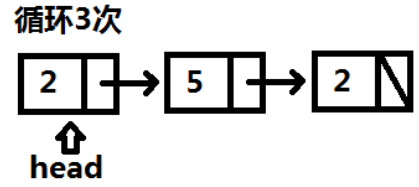


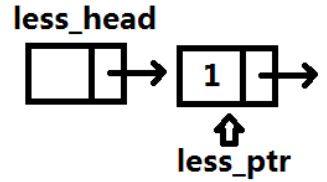


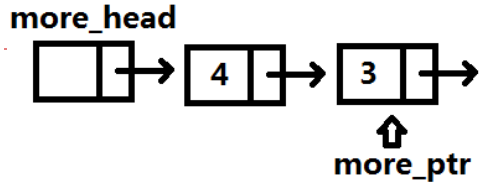


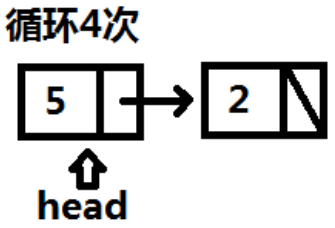


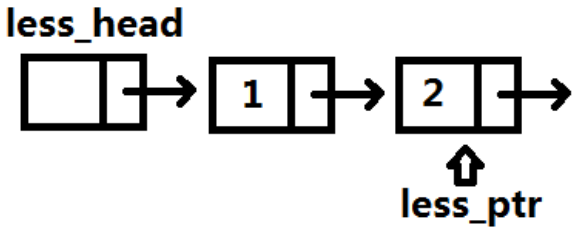


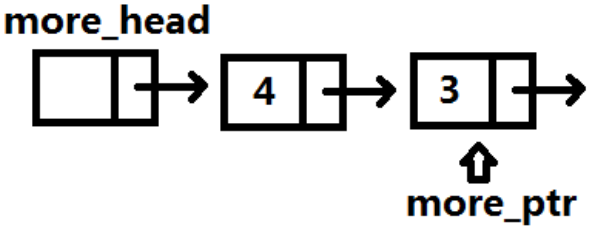




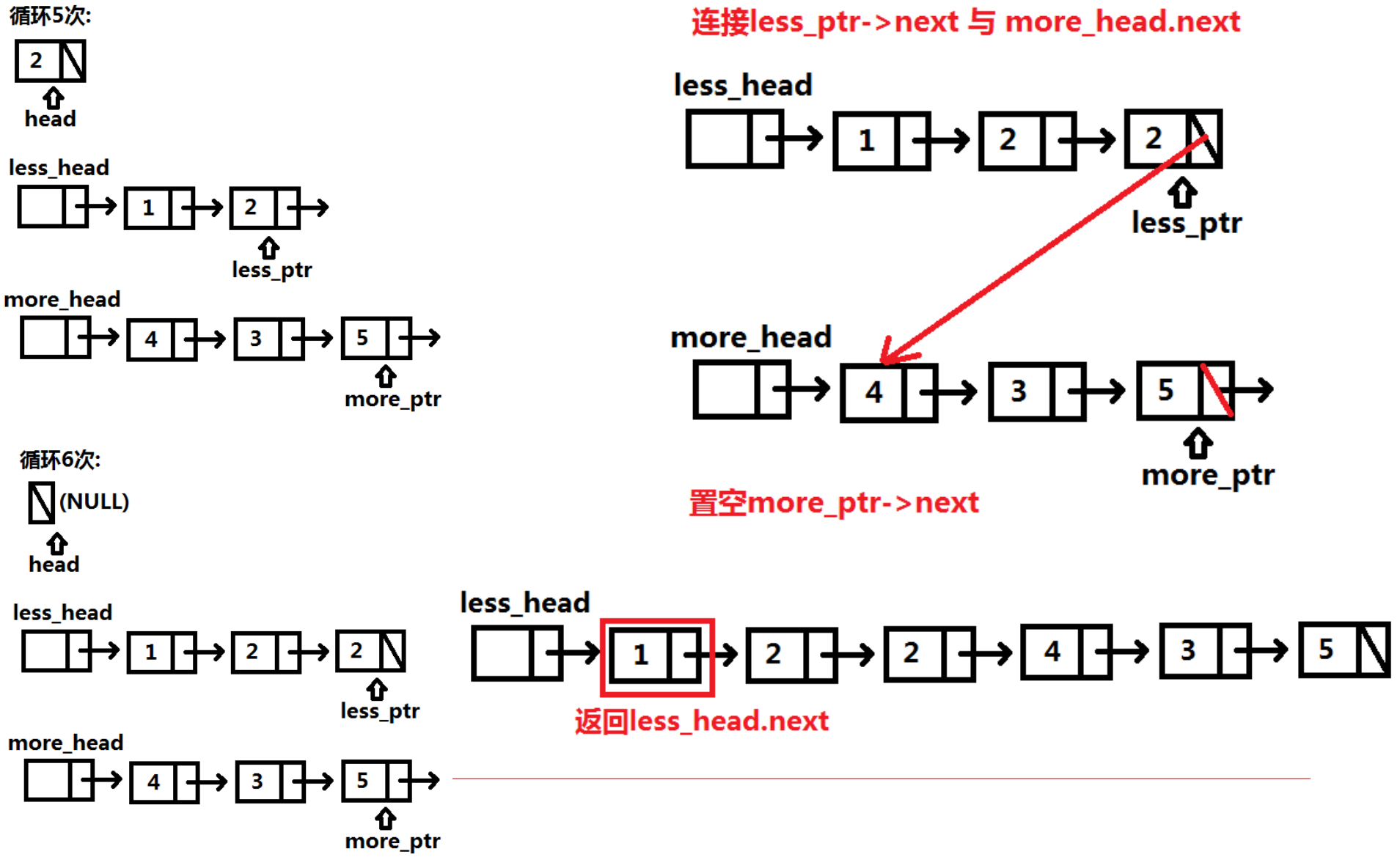








例4:思路，巧用临时头节点



例4:课堂练习

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode less_head(0);
        ListNode more_head(0);
        ListNode *less_ptr = &less_head;
        ListNode *more_ptr = &more_head;
        while(head){
            if (head->val < x){
                less_ptr->next = head;
                    1
            }
            else {
                    2
                more_ptr = head;
            }
            head = head->next;
        }
        3
        more_ptr->next = NULL;
        return less_head.next;
    }
};
```

//设置两个临时的头节点

//对应指针指向这两个头节点

//如果节点值小于x，则将该节点插入less_ptr后

//否则将该节点插入more_ptr后

//遍历链表

//将more_ptr即链表尾节点next置空

//less_head的next节点即为新链表头节点，返回

3分钟时间填写代码，**有问题随时提出！**

例4:实现

```
class Solution {
public:
    ListNode* partition(ListNode* head, int x) {
        ListNode less_head(0);
        ListNode more_head(0);
        ListNode *less_ptr = &less_head;
        ListNode *more_ptr = &more_head;
        while(head){
            if (head->val < x){
                less_ptr->next = head;
                less_ptr = head;
            }
            else {
                more_ptr->next = head;
                more_ptr = head;
            }
            head = head->next;
        }
        less_ptr->next = more_head.next;
        more_ptr->next = NULL;
        return less_head.next;
    }
};
```

//设置两个临时的头节点

//对应指针指向这两个头节点

//如果节点值小于x，则将该节点插入less_ptr后

//链接完成后，less_ptr向后移动，指向head

//否则将该节点插入more_ptr后

//遍历链表

//将less链表尾，与more链表头相连

//将more_ptr即链表尾节点next置空

//less_head的next节点即为新链表头节点，返回

例4:测试与leetcode提交结果

```
int main(){
    ListNode a(1);
    ListNode b(4);
    ListNode c(3);
    ListNode d(2);
    ListNode e(5);
    ListNode f(2);
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    e.next = &f;
    Solution solve;
    ListNode *head = solve.partition(&a, 3);
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    return 0;
}
```

Partition List

Submission Details

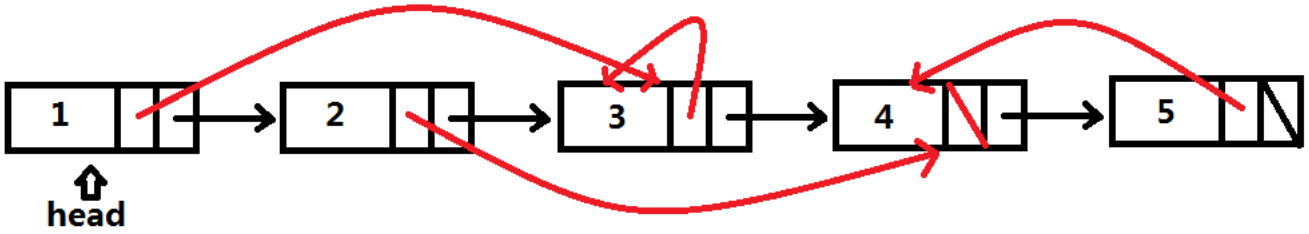
166 / 166 test cases passed. Status: Accepted

Runtime: **6 ms** Submitted: **1 hour, 9 minutes ago**

```
1
2
2
4
3
5
请按任意键继续. . .
```

例5:复杂的链表的深度拷贝

已知一个**复杂**的链表，节点中有一个指向本链表**任意某个**节点的**随机**指针(也可以为空)，求这个链表的**深度拷贝**。




```
struct RandomListNode {
    int label;
    RandomListNode *next, *random;
    RandomListNode(int x) : label(x), next(NULL), random(NULL) {}
};

class Solution {
public:
    RandomListNode *copyRandomList(RandomListNode *head) {
    }
};
```

//带有随机指针的链表节点

//返回是深度拷贝后的链表

//深度拷贝:构造生成一个完全新的链表，即使将原链表毁坏，新链表可独立使用

1分钟理解题目，**有问题随时提出！**

选自 **LeetCode 138. Copy List with Random Pointer**

https://leetcode.com/problems/copy-list-with-random-pointer/description/

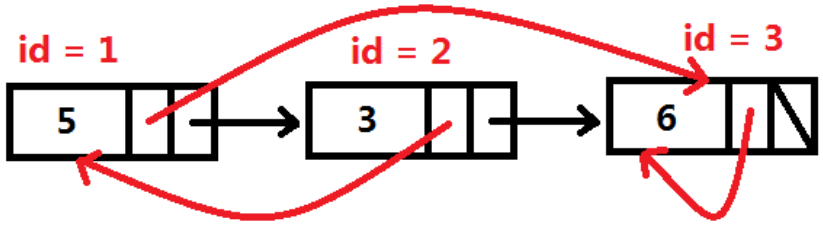
难度:**Hard**

例5:必备知识(STL Map的使用)

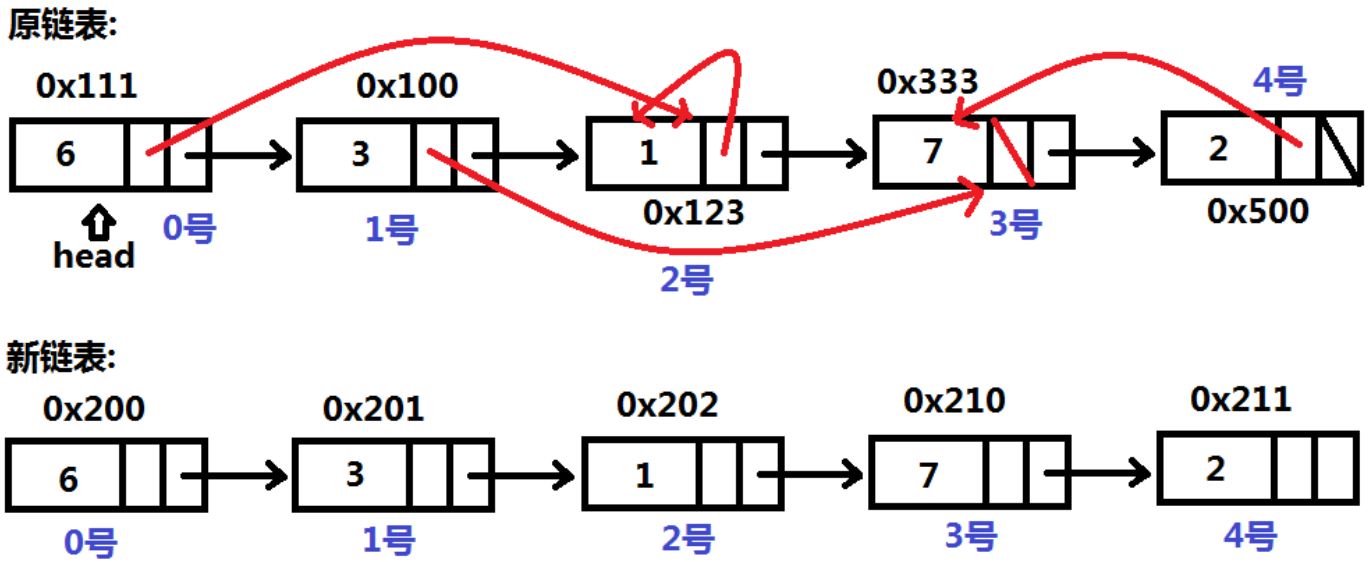
```
a.random id = 3
b.random id = 1
c.random id = 3
请按任意键继续. . .
```

```
#include <stdio.h>
#include <map> //STL map头文件

struct RandomListNode {
    int label;
    RandomListNode *next, *random;
    RandomListNode(int x) : label(x), next(NULL), random(NULL) {}
};

int main(){
    std::map<RandomListNode *, int> node_map;
    RandomListNode a(5);
    RandomListNode b(3);
    RandomListNode c(6);
    a.next = &b;
    b.next = &c;
    a.random = &c;
    b.random = &a;
    c.random = &c;
    node_map[&a] = 1;
    node_map[&b] = 2;
    node_map[&c] = 3;
    printf("a.random id = %d\n", node_map[a.random]);
    printf("b.random id = %d\n", node_map[b.random]);
    printf("c.random id = %d\n", node_map[c.random]);
    return 0;
}
```

//设置一个节点map，key为节点地址，value为整型



例5:思路:节点地址与节点序号对应



Map1 :

原链表节点地址 -> 节点位置(第几个节点)

0x111 -> 0 random 0x123指向 2号节点
0x100 -> 1 random 0x333 指向 3号节点
0x123 -> 2 random 0x123指向 2号节点
0x333 -> 3 random NULL 指向 NULL
0x500 -> 4 random 0x333 指向 3号节点

Map2 :

节点位置(第几个节点) -> 新链表节点地址

0 -> 0x200 random = 2号节点地址(0x202)
1 -> 0x201 random = 3号节点地址(0x210)
2 -> 0x202 random = 2号节点地址(0x202)
3 -> 0x210 random = NULL
4 -> 0x211 random = 3号节点地址(0x210)

例5:课堂练习

```
class Solution {
public:
    RandomListNode *copyRandomList(RandomListNode *head) {
        std::map<RandomListNode *, int> node_map; //地址到节点位置的map
        std::vector<RandomListNode *> node_vec;
        RandomListNode *ptr = head;     //使用vector根据存储节点位置访问地址
        int i = 0;
        while (ptr){ //将新链表节点push入node_vec ,生成了新链表节点位置到地址 的map
            node_vec.push_back(new RandomListNode(ptr->label));
                1
            ptr = ptr->next;
            i++;
        }          //i记录节点位置
        node_vec.push_back(0);
        ptr = head;
        i = 0;          //再次遍历原始列表 连接新链表的next指针、random指针
        while(ptr){
                2
            if (ptr->random){  //当random指针不空时
                int id = node_map[ptr->random]; //根据node_map确认
                    3
            }
            ptr = ptr->next;
            i++;
        }
        return node_vec[0];
    }
};
```

//遍历原始列表

原链表random指针
指向的 位置即id

3分钟时间填写代码，**有问题随时提出！**

例5:实现

```
class Solution {
public:
    RandomListNode *copyRandomList(RandomListNode *head) {
        std::map<RandomListNode *, int> node_map;
        std::vector<RandomListNode *> node_vec;
        RandomListNode *ptr = head;
        int i = 0;
        while (ptr){
            node_vec.push_back(new RandomListNode(ptr->label));
            node_map[ptr] = i;
            ptr = ptr->next;
            i++;
        }
        node_vec.push_back(0);
        ptr = head;
        i = 0;
        while(ptr){
            node_vec[i]->next = node_vec[i+1];
            if (ptr->random){
                int id = node_map[ptr->random];
                node_vec[i]->random = node_vec[id];
            }
            ptr = ptr->next;
            i++;
        }
        return node_vec[0];
    }
};
```

//地址到节点位置的map

//使用vector根据存储节点位置访问地址

//将新链表节点push入node_vec ,生成了新链表节点位置到地址 的map

//记录原始链表地址至节点位置的node_map

//遍历原始列表

//i记录节点位置

//再次遍历原始列表 连接新链表的next指针、random指针

//连接新链表next指针

//当random指针不空时

//根据node_map确认

原链表random指针指向的 位置即id

//连接新链表random指针

例5:测试与leetcode提交结果

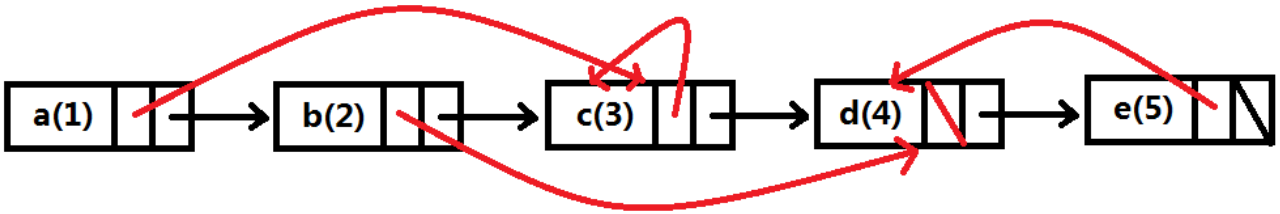
```
int main(){
    RandomListNode a(1);
    RandomListNode b(2);
    RandomListNode c(3);
    RandomListNode d(4);
    RandomListNode e(5);
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
    a.random = &c;
    b.random = &d;
    c.random = &c;
    e.random = &d;
    Solution solve;
    RandomListNode *head = solve.copyRandomList(&a);
    while(head){
        printf("label = %d ", head->label);
        if (head->random){
            printf("rand = %d\n", head->random->label);
        }
        else{
            printf("rand = NULL\n");
        }
        head = head->next;
    }
    return 0;
}
```

Copy List with Random Pointer

Submission Details

12 / 12 test cases passed. Status: Accepted

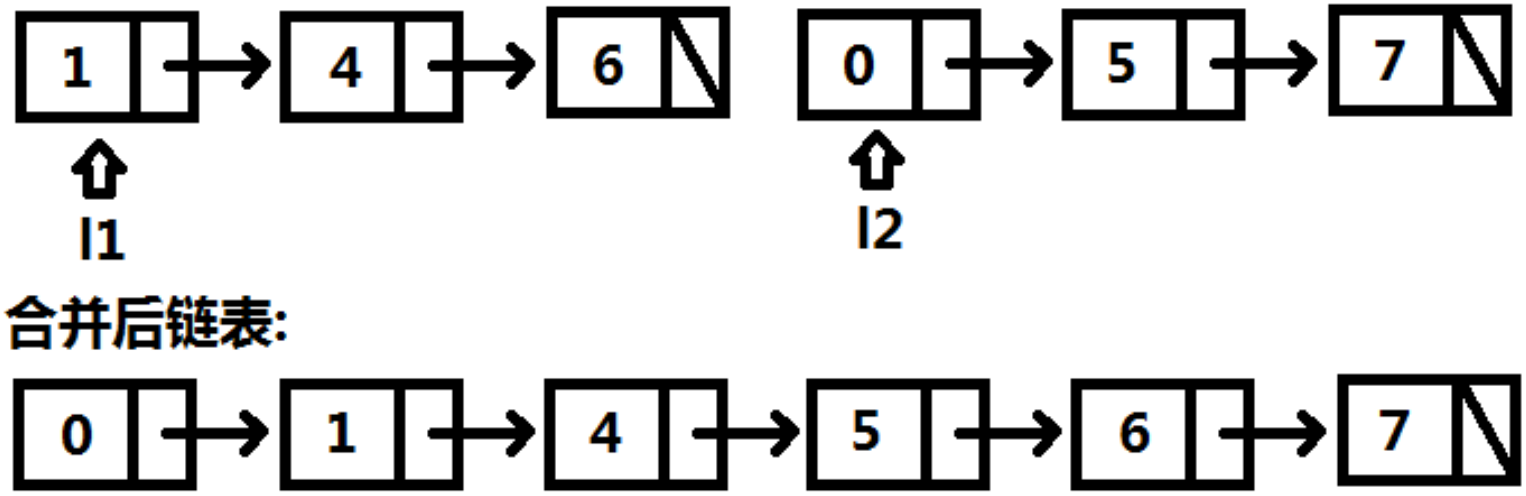
Runtime: **66 ms** Submitted: **0 minutes ago**




```
label = 1 rand = 3
label = 2 rand = 4
label = 3 rand = 3
label = 4 rand = NULL
label = 5 rand = 4
请按任意键继续. . .
```

例6-a:排序链表的合并(2个)

已知两个**已排序**链表头节点指针l1与l2，将这两个链表**合并**，合并后仍为**有序**的，返回合并后的**头节点**。



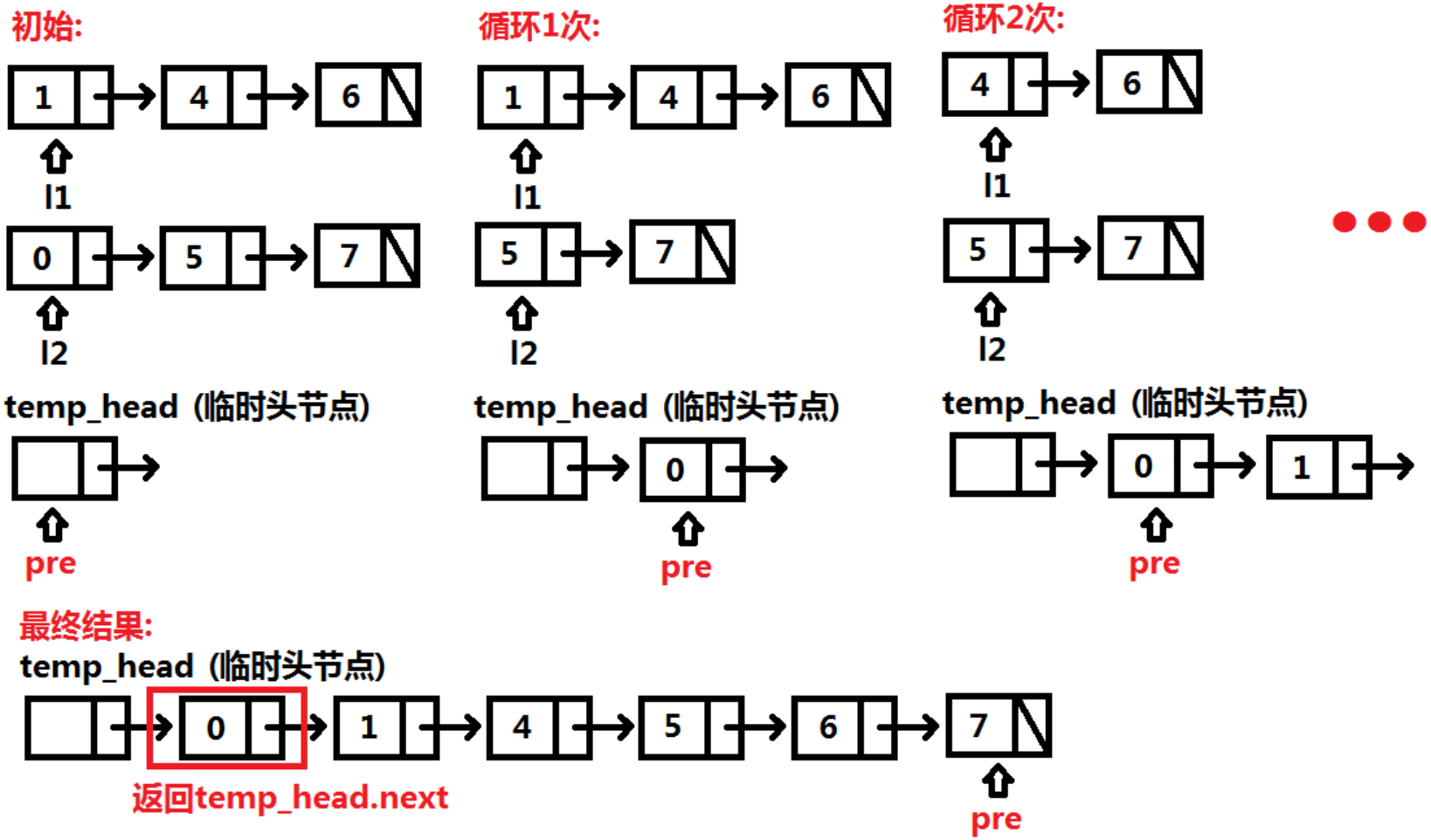
选自 **LeetCode 21. Merge Two Sorted Lists**

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

难度:**Easy**

例6-a:思路

比较l1和l2指向的节点，将**较小的**节点插入到**pre指针**后，并**向前移动**较小节点对应的指针。



例6-a:课堂练习

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode temp_head(0);          //设置临时头节点temp_head
        ListNode *pre = &temp_head;     //使用pre指针指向temp_head
        while (l1 && l2){
            if (l1->val < l2->val){     //l1与l2同时不空时，对它们进行比较
                    1                   //如果l1对应的节点小于l2对应的节点
                l1 = l1->next;
            }
            else{
                    2
                l2 = l2->next;
            }
                3
        }
        if (l1){                        //如果l1有剩余
                4
        }
        if (l2){                        //如果l2有剩余
                5
        }
        return temp_head.next;
    }
};
```

5分钟时间填写代码，**有问题随时提出！**

例6-a:实现

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
        ListNode temp_head(0);          //设置临时头节点temp_head
        ListNode *pre = &temp_head;     //使用pre指针指向temp_head
        while (l1 && l2){
            if (l1->val < l2->val){     //l1与l2同时不空时，对它们进行比较
                pre->next = l1;         //如果l1对应的节点小于l2对应的节点
                l1 = l1->next;
            }                           //将pre与较小的节点进行连接
            else{
                pre->next = l2;
                l2 = l2->next;
            }
            pre = pre->next;            //pre指向新连接的节点
        }
        if (l1){                        //如果l1有剩余
            pre->next = l1;             //将l1接到pre后
        }
        if (l2){                        //如果l2有剩余
            pre->next = l2;             //将l2接到pre后
        }
        return temp_head.next;
    }
};
```

例6-a:测试与leetcode提交结果

```
int main(){
    ListNode a(1);
    ListNode b(4);
    ListNode c(6);
    ListNode d(0);
    ListNode e(5);
    ListNode f(7);
    a.next = &b;
    b.next = &c;
    d.next = &e;
    e.next = &f;
    Solution solve;
    ListNode *head = solve.mergeTwoLists(&a, &d);
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    return 0;
}
```

```
0
1
4
5
6
7
请按任意键继续. . .
```

Merge Two Sorted Lists

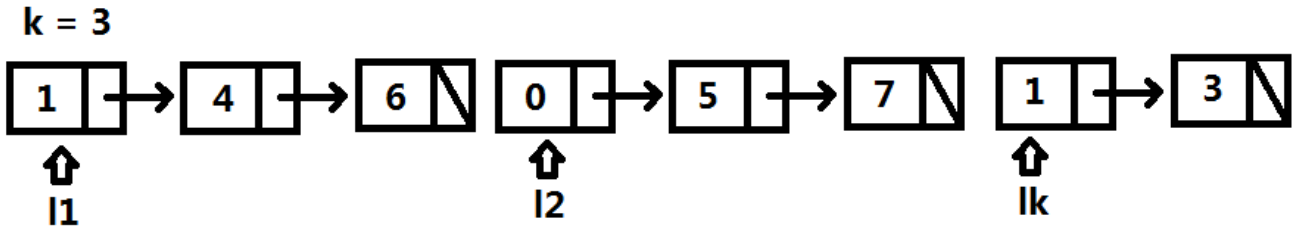
Submission Details

208 / 208 test cases passed. Status: Accepted

Runtime: **9 ms** Submitted: **0 minutes ago**

例6-b:排序链表的合并(多个)

已知k个**已排序**链表头节点指针，将这k个链表**合并**，合并后仍为**有序**的，返回合并后的**头节点**。



合并后链表:


```
class Solution {
public:
    ListNode* mergeKLists(std::vector<ListNode*>& lists) {
    }
};
```

//链表头节点使用vector

//最终返回merge后的有序的链表头

选自 **LeetCode 23. Merge k Sorted Lists**

https://leetcode.com/problems/merge-k-sorted-lists/description/

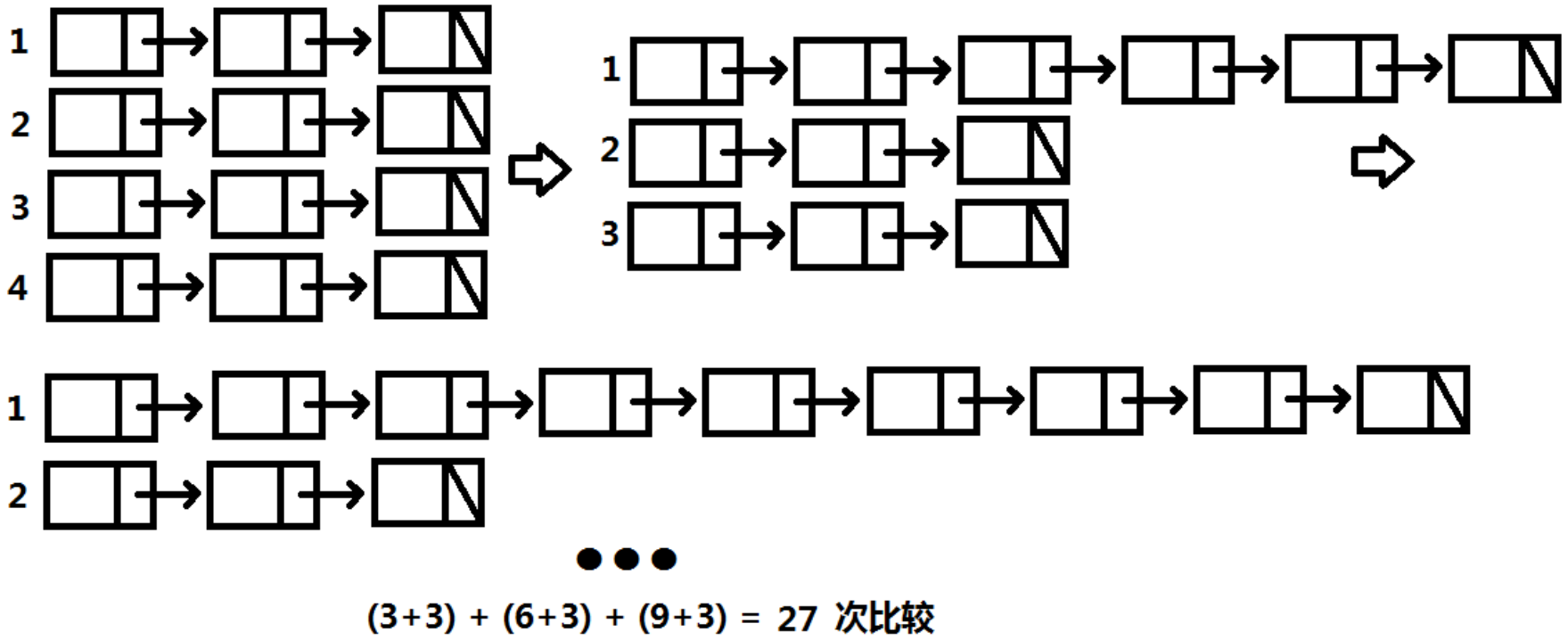
难度:**Hard**

例6-b:方法1思考，暴力合并

方案1:最**普通**的方法，k个链表**按顺序合并**k-1次。

设有**k个链表**，平均每个链表有**n个节点**，时间复杂度:

(n+n)+(2n+n)+((k-1)n+n) = (1+2+…+k-1)n +(k-1)n =(1+2+…+k)n-n=(k^2+k-1)/2 * n = O(k^2*n)



例6-b:方法2思考，排序后相连

方案2:将k*n个节点放到vector中，再将vector**排序**，再将节点**顺序相连**。

设有**k个链表**，平均每个链表有**n个节点**，时间复杂度:

kN*logkN + kN = O(kN*logkN) (比如k=100, n = 10000) logkN = 20， k = 100

```
#include <vector>
#include <algorithm>  //STL 排序算法 std::sort
bool cmp(const ListNode *a, const ListNode *b){
    return a->val < b->val;
}                     //比较函数，对节点进行从小到大的排序
int main(){
    ListNode a(3);
    ListNode b(2);
    ListNode c(5);
    ListNode d(0);
    std::vector<ListNode *> node_vec;
    node_vec.push_back(&a);
    node_vec.push_back(&b);
    node_vec.push_back(&c);
    node_vec.push_back(&d);   //调用排序函数
    std::sort(node_vec.begin(), node_vec.end(), cmp);
    for (int i = 0; i < node_vec.size(); i++){
        printf("%d\n",node_vec[i]->val);
    }
    return 0;
}
```

```
0
2
3
5
请按任意键继续. . .
```

例6-b:方法3思考，分制后相连

方案3:对k个链表进行**分制**，**两两进行合并**。

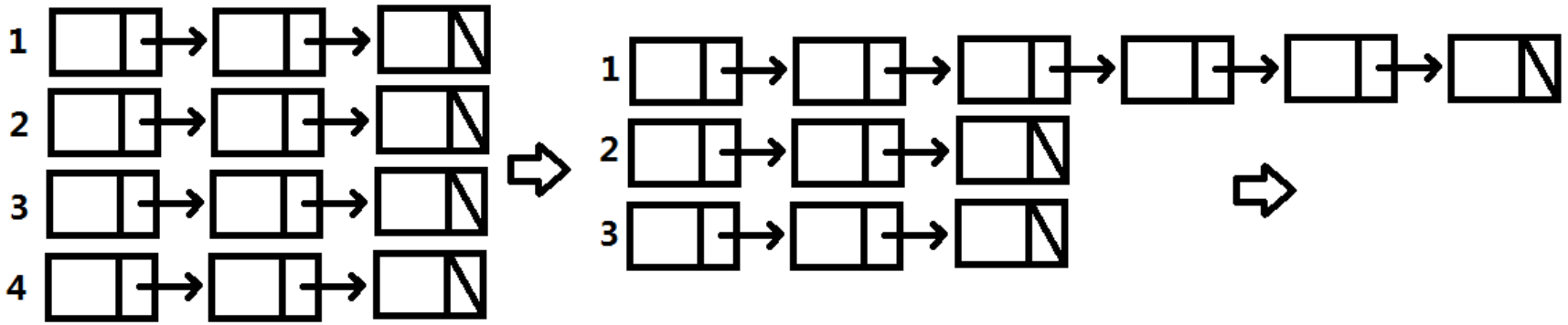
设有**k个链表**，平均每个链表有**n个节点**，时间复杂度:

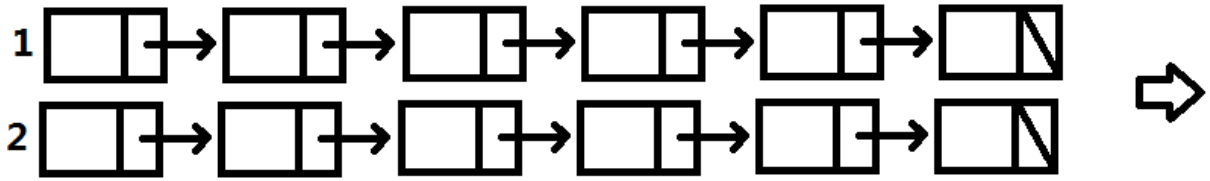
第1轮，进行k/2次，每次处理2n个数字；**第2轮**，进行k/4次，每次处理4n个数字；…；

最后一次，进行k/(2^logk)次，每次处理2^logk*N个值。

2N*k/2 + 4N * k/4 + 8N * k/8 +…+2^logk * N * k/(2^logk)

=Nk + Nk +…+Nk = O(kNlogk)





(3+3) + (3+3) + (6+6) = 18 次比较

例6-b:课堂练习(排序)

```
#include <vector>
#include <algorithm>

bool cmp(const ListNode *a, const ListNode *b){
    return a->val < b->val;
}

class Solution {
public:
    ListNode* mergeKLists(std::vector<ListNode*>& lists) {
        std::vector<ListNode *> node_vec;
        for (int i = 0; i < lists.size(); i++){
            1                                   //遍历k个链表，将节点全部添加至
            while(head){                        node_vec
                2
                head = head->next;
            }
        }
        if (node_vec.size() == 0){
            return NULL;
        }                          //根据节点数值进行排序
        std::sort(node_vec.begin(), node_vec.end(), cmp);
        for (int i = 1; i < node_vec.size(); i++){
            3
        }
        node_vec[node_vec.size()-1]->next = NULL;
        return node_vec[0];
    }
};
```

3分钟时间填写代码，**有问题随时提出！**

例6-b:实现(排序)

```
#include <vector>
#include <algorithm>

bool cmp(const ListNode *a, const ListNode *b){
    return a->val < b->val;
}

class Solution {
public:
    ListNode* mergeKLists(std::vector<ListNode*>& lists) {
        std::vector<ListNode *> node_vec;
        for (int i = 0; i < lists.size(); i++){
            ListNode *head = lists[i];          //遍历k个链表，将节点全部添加至
            while(head){                        node_vec
                node_vec.push_back(head);
                head = head->next;
            }
        }
        if (node_vec.size() == 0){
            return NULL;
        }                       //根据节点数值进行排序
        std::sort(node_vec.begin(), node_vec.end(), cmp);
        for (int i = 1; i < node_vec.size(); i++){          //连接新的链表
            node_vec[i-1]->next = node_vec[i];
        }
        node_vec[node_vec.size()-1]->next = NULL;
        return node_vec[0];
    }
};
```

例6-b:课堂练习(分制)

```
ListNode* mergeKLists(std::vector<ListNode*>& lists) {
    if (lists.size() == 0){
        return NULL;
    }
    if (lists.size() == 1){
        1
    }
    if (lists.size() == 2){
        return mergeTwoLists(lists[0], lists[1]);
    }
    int mid = 2
    std::vector<ListNode*> sub1_lists;
    std::vector<ListNode*> sub2_lists;
    for (int i = 0; i < mid; i++){
        sub1_lists.push_back(lists[i]);
    }
    for (int i = mid; i < lists.size(); i++){
        sub2_lists.push_back(lists[i]);
    }
    ListNode *l1 = 3
    ListNode *l2 = 4
    5
}
```

//如果lists为空，返回NULL

//如果只有两个list，调用两个list merge函数

//拆分lists为两个子lists

3分钟时间填写代码，**有问题随时提出！**

例6-b:实现(分制)

```
ListNode* mergeKLists(std::vector<ListNode*>& lists) {
    if (lists.size() == 0){
        return NULL;
    }
    if (lists.size() == 1){
        return lists[0];
    }
    if (lists.size() == 2){
        return mergeTwoLists(lists[0], lists[1]);
    }
    int mid = lists.size() / 2;
    std::vector<ListNode*> sub1_lists;
    std::vector<ListNode*> sub2_lists;
    for (int i = 0; i < mid; i++){
        sub1_lists.push_back(lists[i]);
    }
    for (int i = mid; i < lists.size(); i++){
        sub2_lists.push_back(lists[i]);
    }
    ListNode *l1 = mergeKLists(sub1_lists);
    ListNode *l2 = mergeKLists(sub2_lists);
    return mergeTwoLists(l1, l2);
}
```

//如果lists为空，返回NULL

//如果只有一个lists，直接返回头指针

//如果只有两个list，调用两个list merge函数

//拆分lists为两个子lists

//分制处理

例6-b:测试与leetcode提交结果

```
int main(){
    ListNode a(1);
    ListNode b(4);
    ListNode c(6);
    ListNode d(0);
    ListNode e(5);
    ListNode f(7);
    ListNode g(2);
    ListNode h(3);
    a.next = &b;
    b.next = &c;
    d.next = &e;
    e.next = &f;
    g.next = &h;
    Solution solve;
    std::vector<ListNode *> lists;
    lists.push_back(&a);
    lists.push_back(&d);
    lists.push_back(&g);
    ListNode *head = solve.mergeKLists(lists);
    while(head){
        printf("%d\n", head->val);
        head = head->next;
    }
    return 0;
}
```

Merge k Sorted Lists **排序算法时间**

Submission Details

130 / 130 test cases passed. Status: Accepted

Runtime: **42 ms** Submitted: **0 minutes ago**

Merge k Sorted Lists **分制算法时间**

Submission Details

130 / 130 test cases passed. Status: Accepted

Runtime: **32 ms** Submitted: **0 minutes ago**

一些建议:关于课程复习与面试准备

1. 在不看回放与复习的前提下，**重新编写**这8道题目进行提交测试。
2. 尽量多的**通过**这些题目。
3. 再看一轮回放，**再编写**一遍题目并提交通过。
4. 纸上写代码，**反复练习**。
5. 将leetcode其他链表相关题目**完成**。

一些建议:关于学习，如何成为一名优秀的研发工程师

1. 掌握一门**编译语言**
2. 掌握**算法与数据结构**
3. 掌握**一门脚本语言**
4. 掌握**开发环境**
5. 丰富其他**前沿知识**

一些建议:关于面试

1. 听清并**搞懂**问题
2. **冷静思考**
3. 努力解决，**不**轻易**放弃**
4. 自信并**谦虚**

一些建议:关于实习

1. **实习是**拿到BAT offer的**捷径**
2. 实习的**岗位有许多**:RD、QA、FE、OP、PM等
3. 尽量在一个实习岗位**坚持**半年
4. 实习还有**不菲的薪水**喔~
5. 关于学校课堂与上课，**自己把握**

结束

非常感谢大家！

林沐