第1,2课 附加补充内容

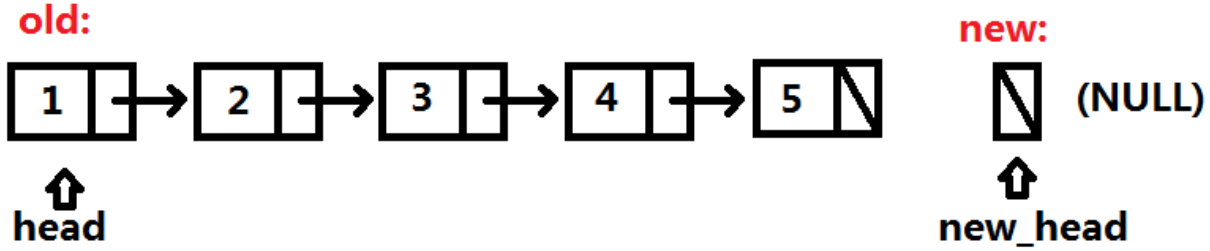
林沐

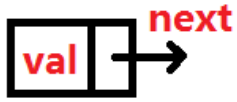
附加补充内容概述

例1:链表逆序拆解详解
例2:链表逆序-头插法
例3:快慢指针求环退出条件分析
例4:使用栈实现队列方法2

例1:链表逆序拆解详解

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






```
old :[1] [2] [3] [4] [5]
new :NULL
```

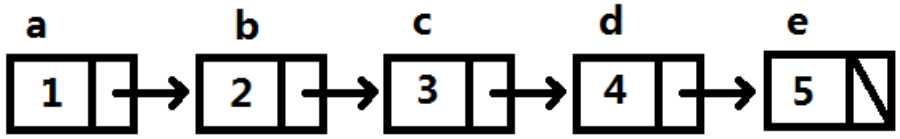
```
#include <stdio.h>

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

void print_list(ListNode *head, const char *list_name){
    printf("%s :", list_name); //打印链表名
    if (!head){
        printf("NULL\n");
        return;
    }                          //如果链表为空，打印NULL，并返回
    while(head){               //遍历链表 并打印链表节点的值
        printf("[%d] ", head->val);
        head = head->next;
    }
    printf("\n");
}
```

例1:初始化一个简单链表

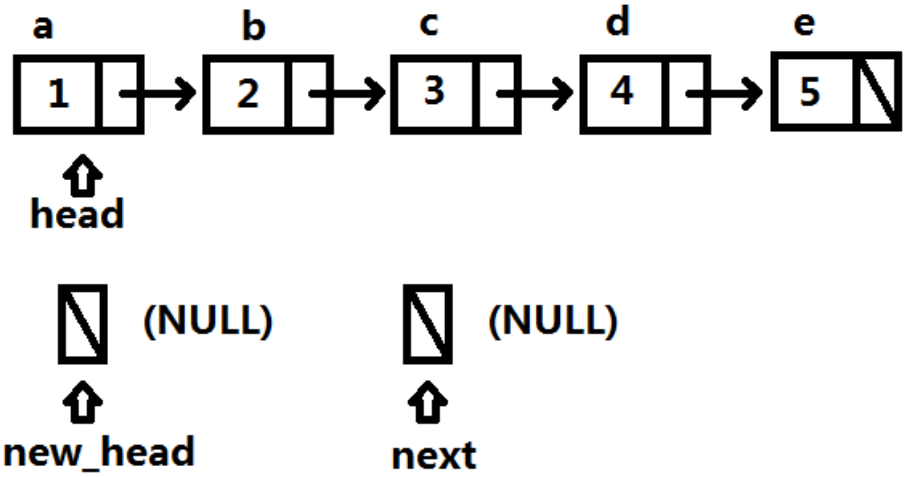
```
int main(){
    ListNode a(1);
    ListNode b(2);
    ListNode c(3);
    ListNode d(4);
    ListNode e(5);
    a.next = &b;
    b.next = &c;
    c.next = &d;
    d.next = &e;
```

1.构造5个节点a,b,c,d,e；并对它们的val做初始化。

2.将a,b,c,d,e 5个节点链接在一起。




```
    ListNode *head = &a;
    ListNode *new_head = NULL;
    ListNode *next = NULL;
    print_list(head, "old");
    print_list(new_head, "new");
```

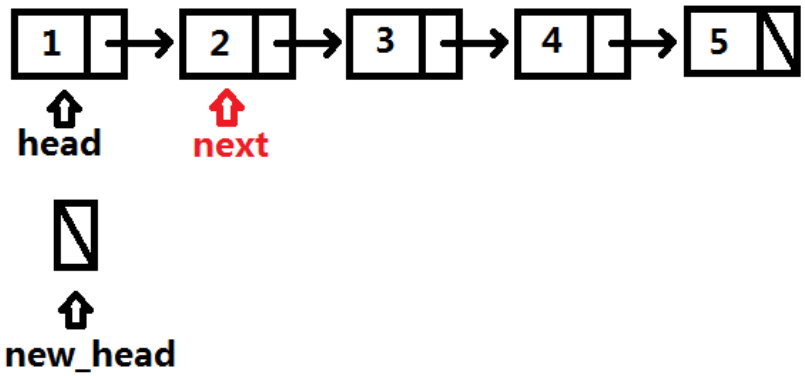



```
old :[1] [2] [3] [4] [5]
new :NULL
```

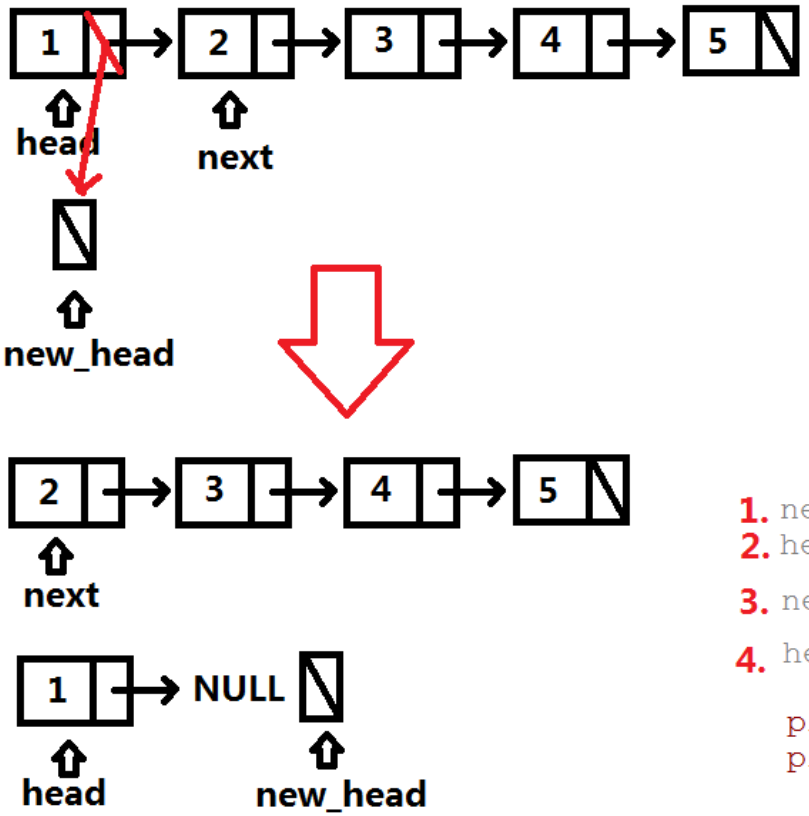
例1:逆置第1个节点

第一组代码:

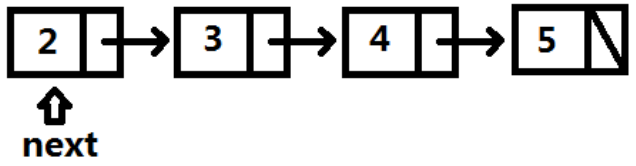
1. next = head->next;

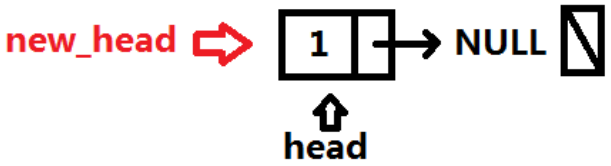


2.head->next = new_head;



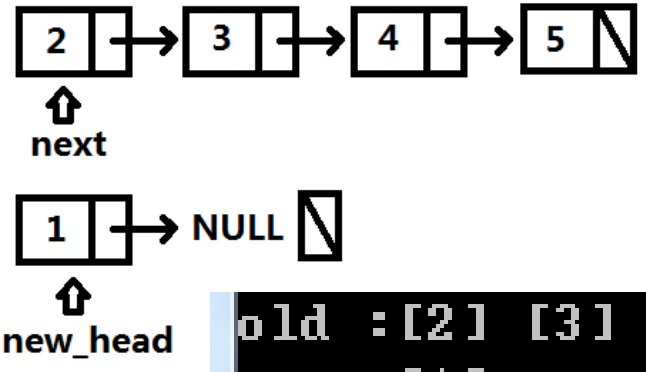
3.new_head = head;





head

4.head = next;



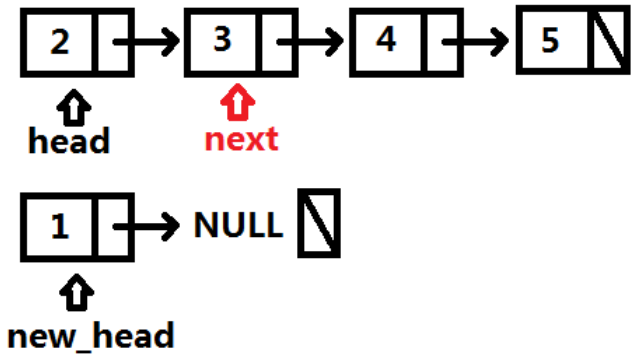

```
old :[2] [3] [4] [5]
new :[1]
```

```
1. next = head->next;
2. head->next = new_head;
3. new_head = head;
4. head = next;

   print_list(head, "old");
   print_list(new_head, "new");
```

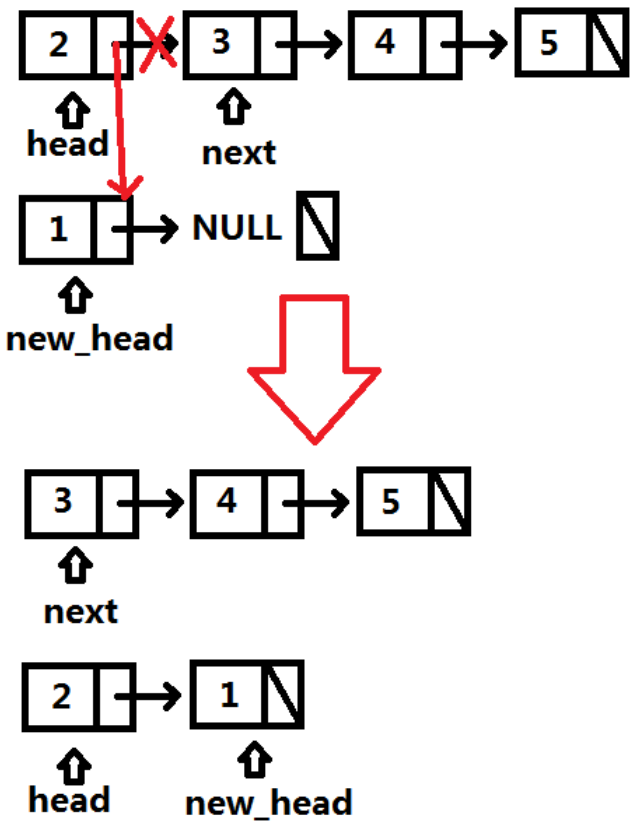
例1:逆置第2个节点

第2组代码:

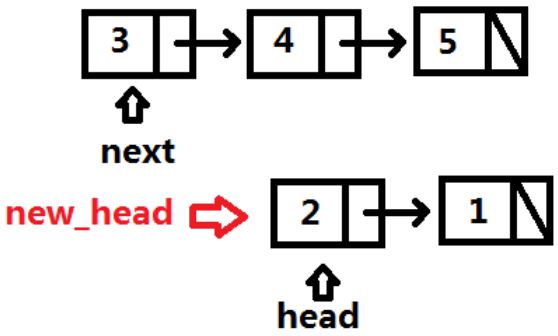
1. next = head->next;



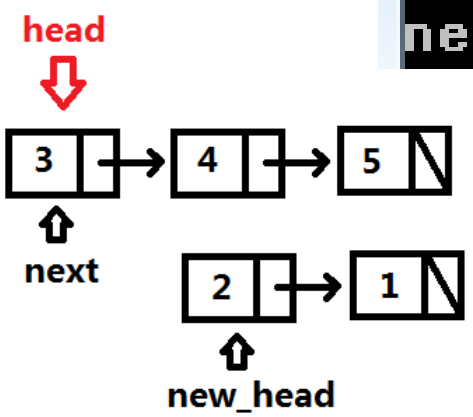
2.head->next = new_head;



3.new_head = head;



4.head = next;



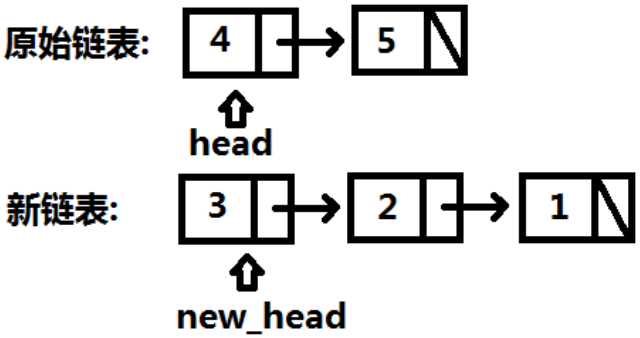

```
old :[3] [4] [5]
new :[2] [1]
```

1. `next = head->next;`
2. `head->next = new_head;`
3. `new_head = head;`
4. `head = next;`

```
print_list(head, "old");
print_list(new_head, "new");
```

例1:逆置第3,4,5节点

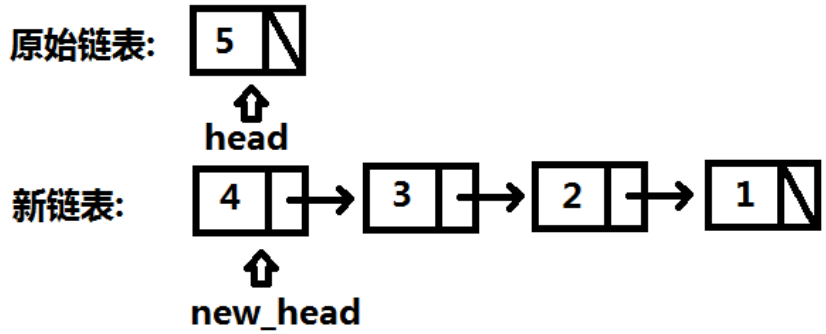
第3组代码:




```
next = head->next;
head->next = new_head;
new_head = head;
head = next;
print_list(head, "old");
print_list(new_head, "new");
```

```
old :[4] [5]
new :[3] [2] [1]
```

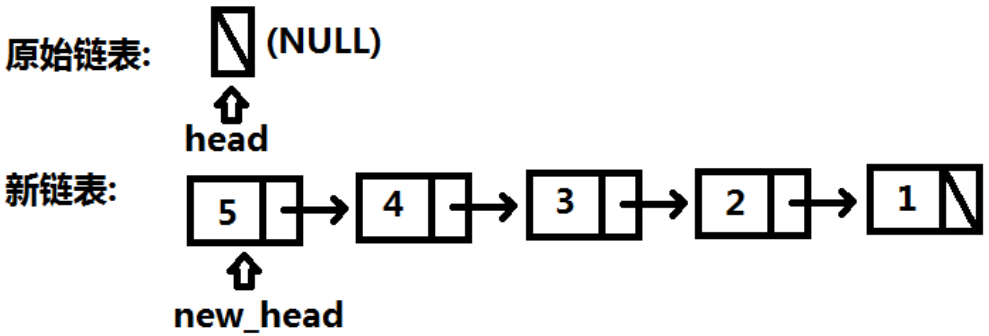
第4组代码:




```
next = head->next;
head->next = new_head;
new_head = head;
head = next;
print_list(head, "old");
print_list(new_head, "new");
```

```
old :[5]
new :[4] [3] [2] [1]
```

第5组代码:




```
next = head->next;
head->next = new_head;
new_head = head;
head = next;
print_list(head, "old");
print_list(new_head, "new");
```

```
old :NULL
new :[5] [4] [3] [2] [1]
```

例1:将代码放在循环里写

```
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;

    ListNode *head = &a;
    ListNode *new_head = NULL;
    ListNode *next = NULL;
    print_list(head, "old");
    print_list(new_head, "new");

    for (int i = 0; i < 5; i++){
        next = head->next;
        head->next = new_head;
        new_head = head;
        head = next;
        print_list(head, "old");
        print_list(new_head, "new");
    }
    return 0;
}
```

//把这段代码放在循环里写

//用i循环5次

```
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;

    ListNode *head = &a;
    ListNode *new_head = NULL;
    ListNode *next = NULL;
    print_list(head, "old");
    print_list(new_head, "new");

    while(head){
        next = head->next;
        head->next = new_head;
        new_head = head;
        head = next;
        print_list(head, "old");
        print_list(new_head, "new");
    }
    return 0;
}
```

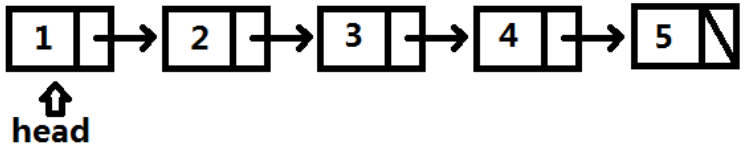
//不再利用i循环，利用head循环

```
old :[1] [2] [3] [4] [5]
new :NULL
old :[2] [3] [4] [5]
new :[1]
old :[3] [4] [5]
new :[2] [1]
old :[4] [5]
new :[3] [2] [1]
old :[5]
new :[4] [3] [2] [1]
old :NULL
new :[5] [4] [3] [2] [1]
请按任意键继续. . .
```

例2:链表逆序-头插法

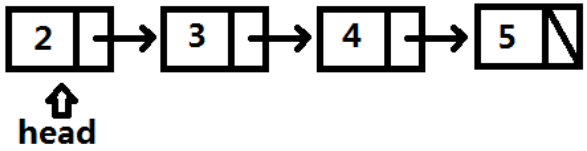
设置一个临时头节点temp_head，利用head指针遍历链表，
每遍历一个节点即将该节点插入到temp_head后。

初始状态，待插入1号节点:



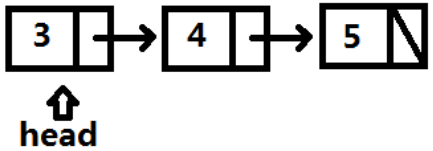
temp_head

完成1号节点插入，待插入2号节点:

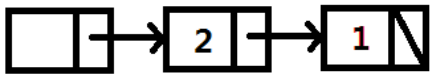


temp_head

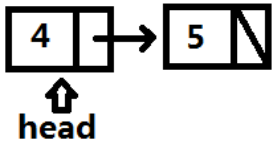
完成2号节点插入，待插入3号节点:



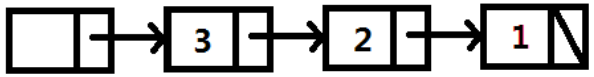
temp_head



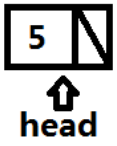
完成3号节点插入，待插入4号节点:



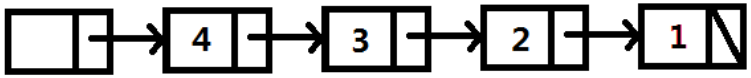
temp_head



完成4号节点插入，待插入5号节点:



temp_head

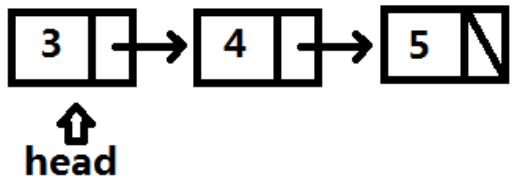


完成5号节点插入，所有节点均完成遍历，head指向了空

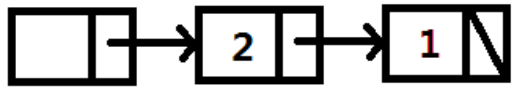
temp_head

例2:链表逆序-头插法

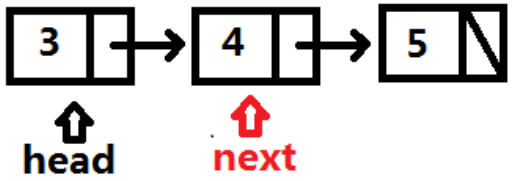
插入head指向的某一节点:



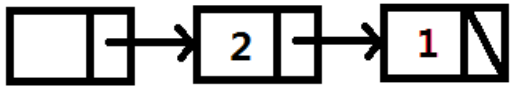
temp_head



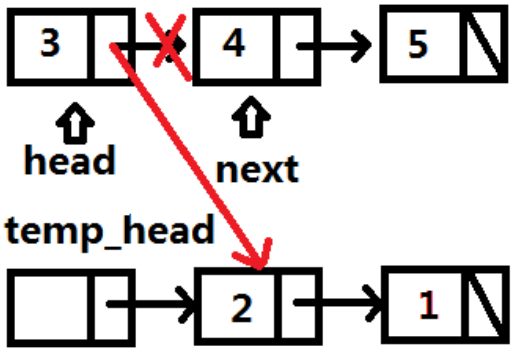
1. 备份 next = head->next;



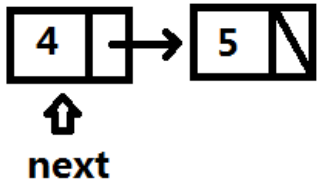
temp_head



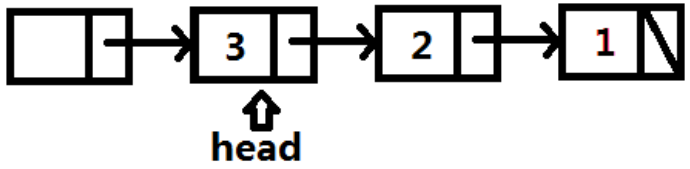
2.修改head->next, head->next = temp_head.next;



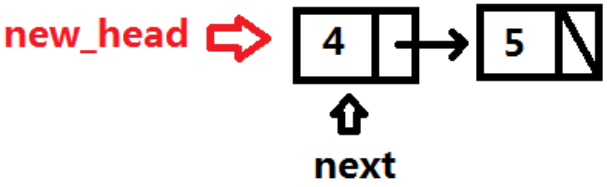
3.修改temp_head.next , temp_head.next = head



temp_head



4.移动head , head = next;



temp_head

例2:链表逆序-两种方法实现的比较

头插法:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode temp_head(0);
        while(head){
            ListNode *next = head->next;
            head->next = temp_head.next;
            temp_head.next = head;
            head = next;
        }
        return temp_head.next;
    }
};
```

就地逆置法:

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode *new_head = NULL;
        while(head){
            ListNode *next = head->next;
            head->next = new_head;
            new_head = head;
            head = next;
        }
        return new_head;
    }
};
```

例3:快慢指针求环退出条件分析

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *fast = head;
        ListNode *slow = head;
        ListNode *meet = NULL;
        while(fast){
            slow = slow->next;
            fast = fast->next;
            if (!fast){
                return NULL;
            }
            fast = fast->next;
            if (fast == slow){
                meet = fast;
                break;
            }
        }
        if (meet == NULL){
            return NULL;
        }
        while(head && meet){
            if (head == meet){
                return head;
            }
            head = head->next;
            meet = meet->next;
        }
        return NULL;
    }
};
```

//快慢指针

//相遇的节点

//链表无环且，且节点个数为偶数个，这里跳出循环

//slow与fast先各走一步

//链表无环，且节点个数为奇数个，这里返回

//如果fast遇到链表尾，则返回NULL

//fast再走1步

//fast与slow相遇，记录相遇位置

其实这段if判断确实没什么用!

//如果没有相遇，则证明无环

因为这里已经判断了!

当head与meet相遇，说明遇到环的起始位置

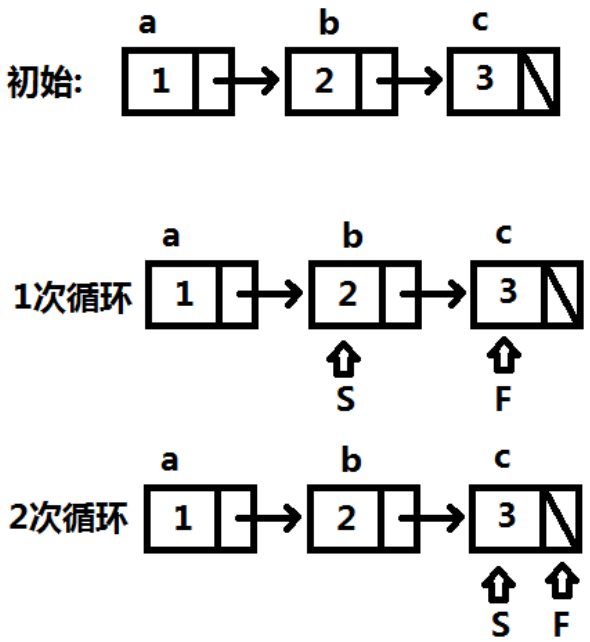
//head与meet每次走1步

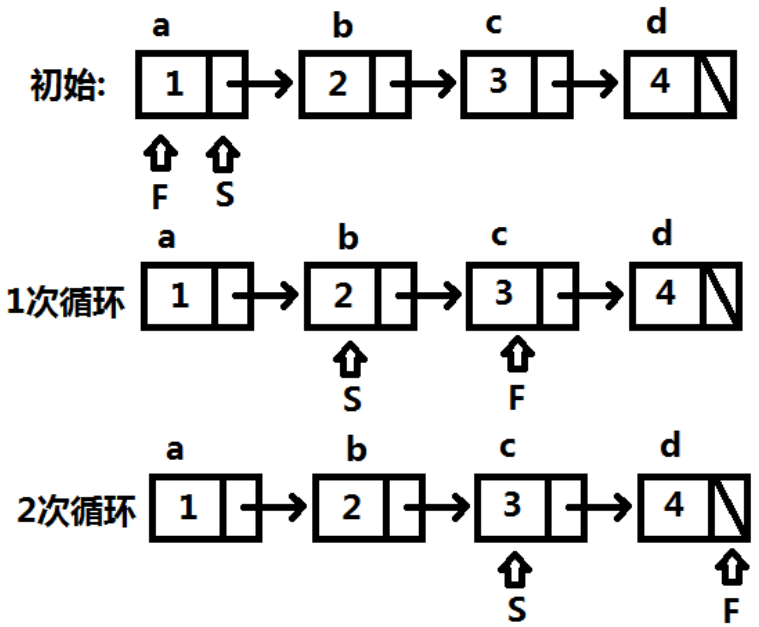
例3:分析

```
#include <stdio.h>

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

int check_nodes(ListNode *head) {
    ListNode *fast = head;
    ListNode *slow = head;
    while(fast){
        slow = slow->next;
        fast = fast->next;
        if (!fast){
            return 1; //这里是节点奇数个时退出的地方
        }
        fast = fast->next;
    }
    return 0; //这里是节点个数是偶数个时推出的地方
}
```

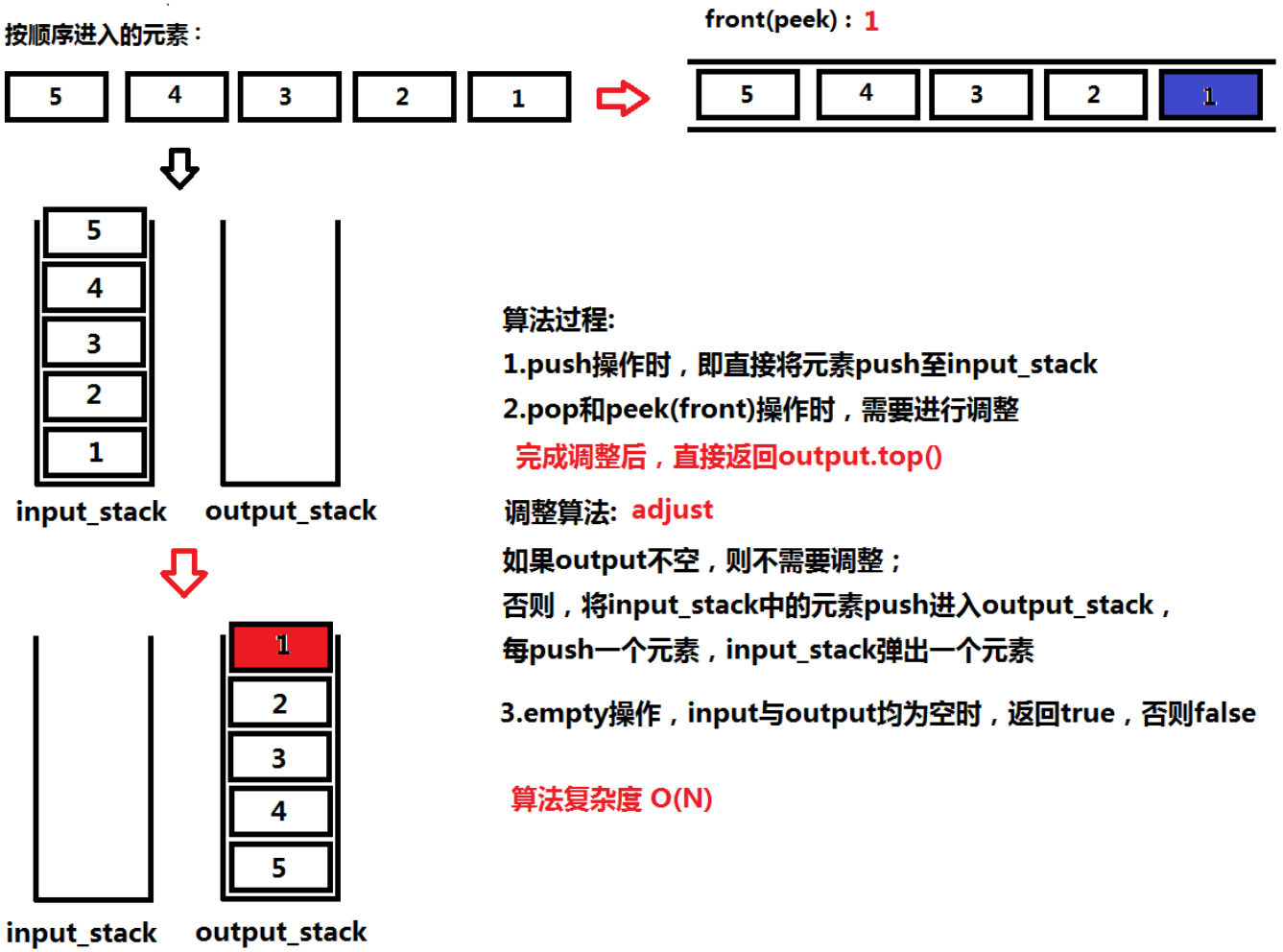
```
int main(){
    ListNode a(1);
    ListNode b(2);
    ListNode c(3);
    a.next = &b;
    b.next = &c;

    ListNode d(1);
    ListNode e(2);
    ListNode f(3);
    ListNode g(4);
    d.next = &e;
    e.next = &f;
    f.next = &g;
    printf("a list %d\n", check_nodes(&a));
    printf("d list %d\n", check_nodes(&d));
    return 0;
}
```

```
a list 1
d list 0
```





例4:使用栈实现队列(方法2双栈法)



算法过程:

1.push操作时，即直接将元素push至input_stack

2.pop和peek(front)操作时，需要进行调整

完成调整后，直接返回output.top()

调整算法: adjust

如果output不空，则不需要调整；

否则，将input_stack中的元素push进入output_stack，

每push一个元素，input_stack弹出一个元素

3.empty操作，input与output均为空时，返回true，否则false

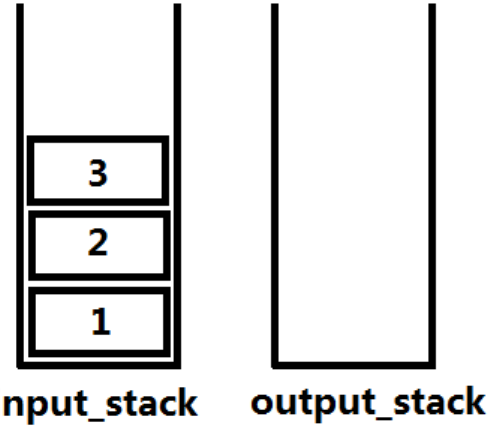
算法复杂度 O(N)

例4:使用栈实现队列(方法2双栈法)

按顺序进入的元素：

1. Q.push(1); input_stack.push(1);
2. Q.push(2); input_stack.push(2);
3. Q.push(3); input_stack.push(3);
4. Q.front(); -> 1 (adjust) output.top()
5. Q.push(4); input_stack.push(4);
6. Q.pop(); (adjust) output_stack.pop();
7. Q.front(); -> 2 (adjust) output.top()
8. Q.pop(); (adjust) output_stack.pop();
9. Q.push(5); input_stack.push(5);
10. Q.front(); -> 3 (adjust)
11. Q.pop(); (adjust) output_stack.pop();
12. Q.pop(); (adjust) output_stack.pop();

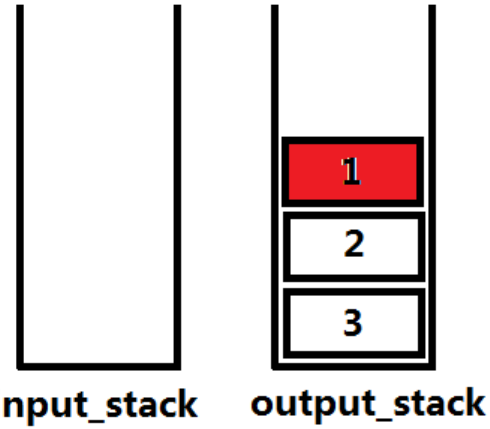
图1:



1. Q.push(1);
2. Q.push(2);
3. Q.push(3);

input_stack.push(1);
input_stack.push(2);
input_stack.push(3);

图2: adjust

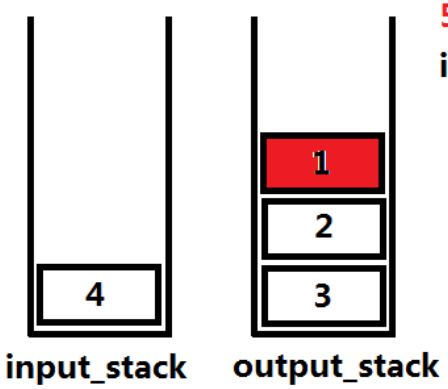


4. Q.front(); -> 1

(adjust) output.top()

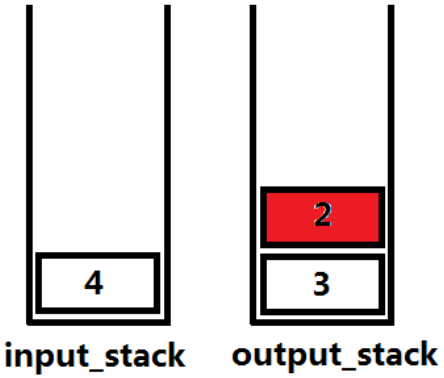
例4:使用栈实现队列(方法2双栈法)

图3:



5. Q.push(4);
input_stack.push(4);

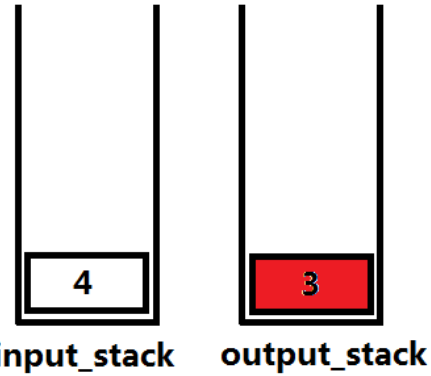
图4:



6. Q.pop();
(adjust)
由于output_stack不空，则无需调整(adjust)
output_stack.pop();

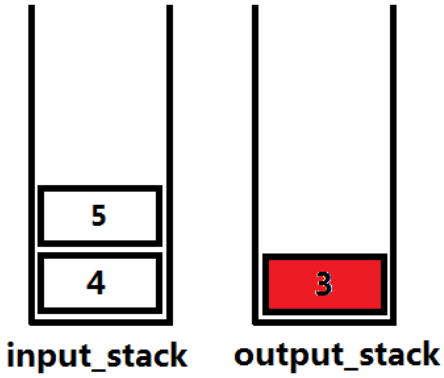
7. Q.front(); -> 2
(adjust) 无需调整
output.top()

图5:



8. Q.pop();
(adjust) output_stack.pop();

图6:

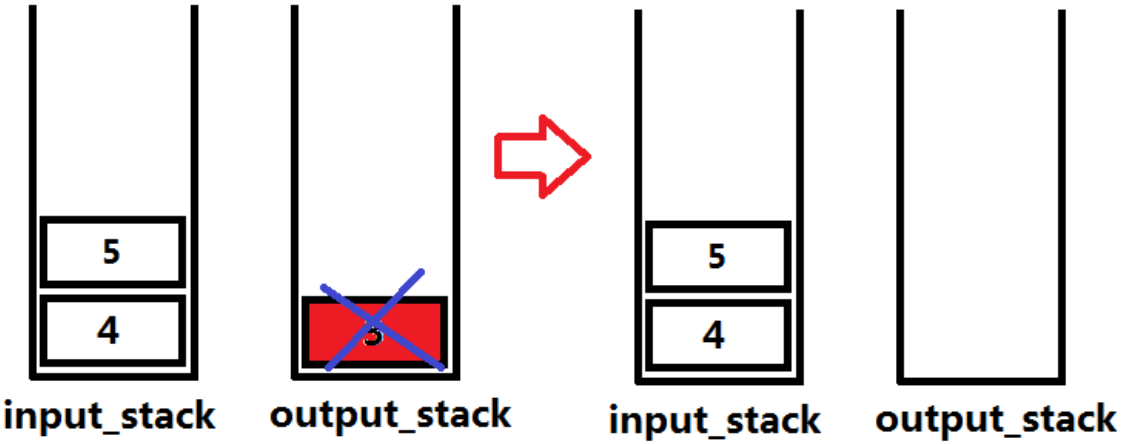


9. Q.push(5);
input_stack.push(5);

10. Q.front(); -> 3
(adjust)
由于output_stack不空，则无需调整(adjust)

例4:使用栈实现队列(方法2双栈法)

图7:

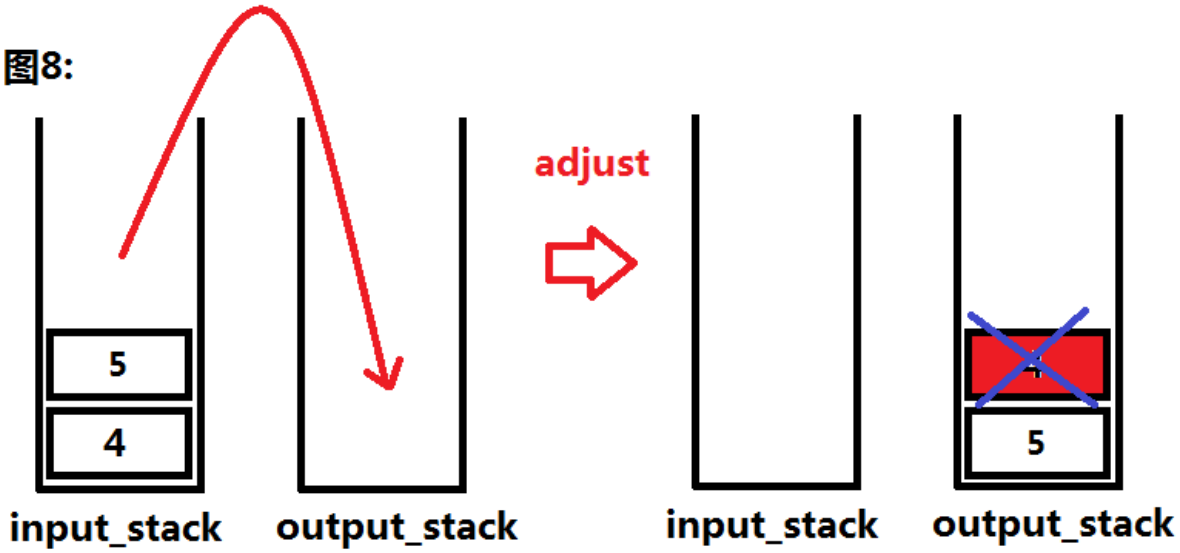


11. Q.pop();
(adjust)

由于output_stack不空，则无需调整(adjust)

output_stack.pop();

图8:



12. Q.pop();
(adjust)

output_stack.pop();

例4:双栈法(课堂练习)

```
#include <stack>
class MyQueue {
public:
    MyQueue() {
    }
    void push(int x) {
        1                          //直接将x push进入input
    }
    int pop() {
        2                          //调整再进行pop
        int x = _output.top();
        _output.pop();
        return x;
    }
    int peek() {                   //调整，并返回output_stack.top()
        adjust();
        return _output.top();
    }
    bool empty() {
        return 3
    }
private:
    void adjust(){
        if (  4  ){ //当??情况，无需调整
            return;
        }                          //调整的过程
        while(!_input.empty()){
            5
            _input.pop();
        }
    }
    std::stack<int> _input;
    std::stack<int> _output;
};
```

3分钟时间填写代码，

有问题随时提出！

例4:双栈法(实现)

```
#include <stack>
class MyQueue {
public:
    MyQueue() {
    }
    void push(int x) {
        _input.push(x);          //直接将x push进入input
    }
    int pop() {
        adjust();                //调整再进行pop
        int x = _output.top();
        _output.pop();
        return x;
    }
    int peek() {                 //调整，并返回output_stack.top()
        adjust();
        return _output.top();
    }
    bool empty() {               //当input_stack与output_stack同时为空时，才返回true
        return _input.empty() && _output.empty();
    }
private:
    void adjust(){
        if (!_output.empty()) {  //当output_stack不空的时候
            return;
        }                        //调整的过程
        while(!_input.empty()){
            _output.push(_input.top());
            _input.pop();        //将input_stack中的每个元素均push
        }                        进入output_stack，每push一个input弹出一个
    }
    std::stack<int> _input;
    std::stack<int> _output;
};
```

关于算法思维修炼的各个阶段

阶段0 : 初学乍练,不足挂齿

推荐《算法导论》、《算法竞赛入门经典》

阶段1 : 粗懂皮毛,半生不熟

http://train.usaco.org/usacogate

https://leetcode.com/

阶段2 : 已有小成,融会贯通

阶段3 : 炉火纯青,出类拔萃

阶段4 : 登峰造极,举世无双

结束

非常感谢大家！

林沐