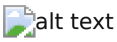
□转链表

 alt text

□转链表迭代做法

通过创建一个新的节点**prev**，将旧链表从前往后遍历

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

□转链表头插法做法

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

□转链表递归做法

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
    if(head == NULL || head->next == NULL) return head;
    ListNode* tail = head->next;
    ListNode* newhead = reverseList(head->next);   //□reverseList
    head->next = tail->next;
```

（head->next）看成一个新的链表，其头节

```
    tail->next = head;      点为newhead
    return newhead;
    }
};
```

判断环形链表

快慢指针法

- 两个指针来判断
- 一个指针跑得快，一个指针跑得慢
- 若不是环形结构，跑的慢的永远追不上跑得快的，因此一定会返回false
- 若是环形结构，跑得慢的一定会遇到跑得快的，因此一定会返回true

根据上述思考来设计代码

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *p = head , *q = head;
        while(q && q->next)   //要加上q->next不指向null的条件
        {                       否则当链表只有一个节点时便会出错
            p = p->next;
            q = q->next->next;
            if(p == q) return true;
        }
        return false;
    }
};
```

快乐数

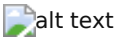
（使用快慢指针思想来做这道题）

 alt text

- 情况1：一直在1中死循环
- 情况2：一直在某些数据中死循环，但到达不了1 因此，我们可以使用快慢指针的思想来解题

代码：

```
class Solution {
public:
    int get(int n)
    {
        int y=0,d;
        while(n)
        {
        d = n % 10;
        y += d * d;
        n = n / 10;
        }
        return y;
```



```
    }
    bool isHappy(int n) {
        int p = n,q = n;
        while(q!=1)  // q是快指针，因此使用q来判定是否循环（q若可以等于
        {                1，则p也一定可以等于1）
            p=get(p);
            q=get(get(q));
            if(p==q && q!=1) // p和q相等但不是在1中死循环
            return false;
        }
        return true;
    }
};
```

双指针等距移动法

旋转链表

alt text

个人解法：

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(head == NULL) return head;
    int len = 0;      //链表长度
    ListNode *p = head ;
    ListNode *q = head,*temp,*newhead;
    while (q)
    {
        q = q->next;
        len+=1;
    }
    int x = k % len;
    if(x == 0) return head;
    for(int i = 1;i < len-x ; i++)//分割成俩链表
        p = p->next;          //遍历至前一链表末端
    temp = p->next;         //记录下一链表的头节点
    newhead = p->next;
    p->next = NULL;       //将前一链表末端指null
    while(temp->next != NULL) //遍历后一链表至末端
    temp = temp->next;
    temp->next = head;
    return newhead;
    }
};
```

双指针移动法：要判断特殊情况

```
class Solution {
public:
    ListNode* rotateRight(ListNode* head, int k) {
        if(head == NULL)   return head;
        ListNode *p = head , *q = head,*temp = head;
        int len = 0;
        while(head)
        {
            head = head->next;
            len+=1;
        }
        k %= len;
        if(k==0) return temp;
        for(int i = 0;i <= k ;i++)
        q = q->next;
        while(q)
        {
            p = p->next;
            q = q->next;
        }
        q = p->next;
        p->next = NULL;
        p = q;
        while(p->next)
        p = p->next;
        p->next = temp;
        return q;
    }
};
```

□□□□□□□□N□□□

 alt text □□□□□

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {

    ListNode *p = head,*q = head;
    ListNode new_head;
    new_head.next = head;
    int len = 0;
    while(q)
    {
        q = q->next;
        len +=1;
    }
    if(n == len)
    {
    new_head.next = head->next;
```

```
    return new_head.next;
    }
    for(int i = 1;i < len-n ;i++)
        p = p->next;
        p->next = p->next->next;
        return head;
    }
};
```

需要考虑边界条件，当要删除第一个节点时，便不能**return head**，而需要找到虚拟头节点进行**return**，同时运行速度也不快

双指针移动法：（使虚拟头节点）

- 使两个指针，一快一慢，并且都指向虚拟头节点
- 让快的指针先往后走$n+1$位
- 然后两个指针一起往后走，当快指针走到null时，慢指针走到了待删除节点的前一位

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode new_head , *p = &new_head , *q = p;
        new_head.next = head;
        for(int i = 1; i <= n+1;i++)
        q = q->next;
        while (q)
        {
            p = p->next;
            q = q->next;
        }
        p->next = p->next->next;
        return new_head.next;
    }
};
```

好处：

1. 不需要考虑边界条件，因为有了虚拟头节点的存在
2. 运行速度更快

环形链表II

alt text 这道题的解法非常巧妙 同样使用快慢指针，得出其路程之间的关系

- 设定两个指针，一个一次走两步，一个一次走一步（设头节点距环入口为的距离为a）
- 如果有环，他俩必定相遇
- 通过数学关系得出相遇的位置到环入口的距离等于头节点到环入口的距离
    - 设环长度为x，在慢指针刚到达环入口时，快指针领先他a的距离
    - 快指针与慢指针距离为$x-a$
    - 快指针追上慢指针需要再走$x-a$次
    - 慢指针在此过程中走了$x-a$步，相遇时的位置距离环入口为$x-（x-a）=a$的距离

    - 相遇位置与头节点距环入口距离相同  alt text

- 将其中一个指针指向头节点，让它们以相同速度走，相遇位置即环入口

附上代码：

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode *q =head , *p = head;
        while(p && p->next)
        {
            q = q->next;
            p = p->next->next;
            if(p==q)
            break;
        }
        if (p == NULL || p->next == NULL)
        return NULL;
        q = head;
        while(p != q)
        {
            p = p->next;
            q = q->next;
        }
        return q;
    }
};
```