1. 输出单链表倒数第 K 个节点

1.1 问题描述

题目：输入一个单链表，输出此链表中的倒数第 K 个节点。（去除头结点，节点计数从 1 开始）。

1.2 两次遍历法

1.2.1 解题思想

（1）遍历单链表，遍历同时得出链表长度 N 。

（2）再次从头遍历，访问至第 N - K 个节点为所求节点。

1.2.2 图解过程

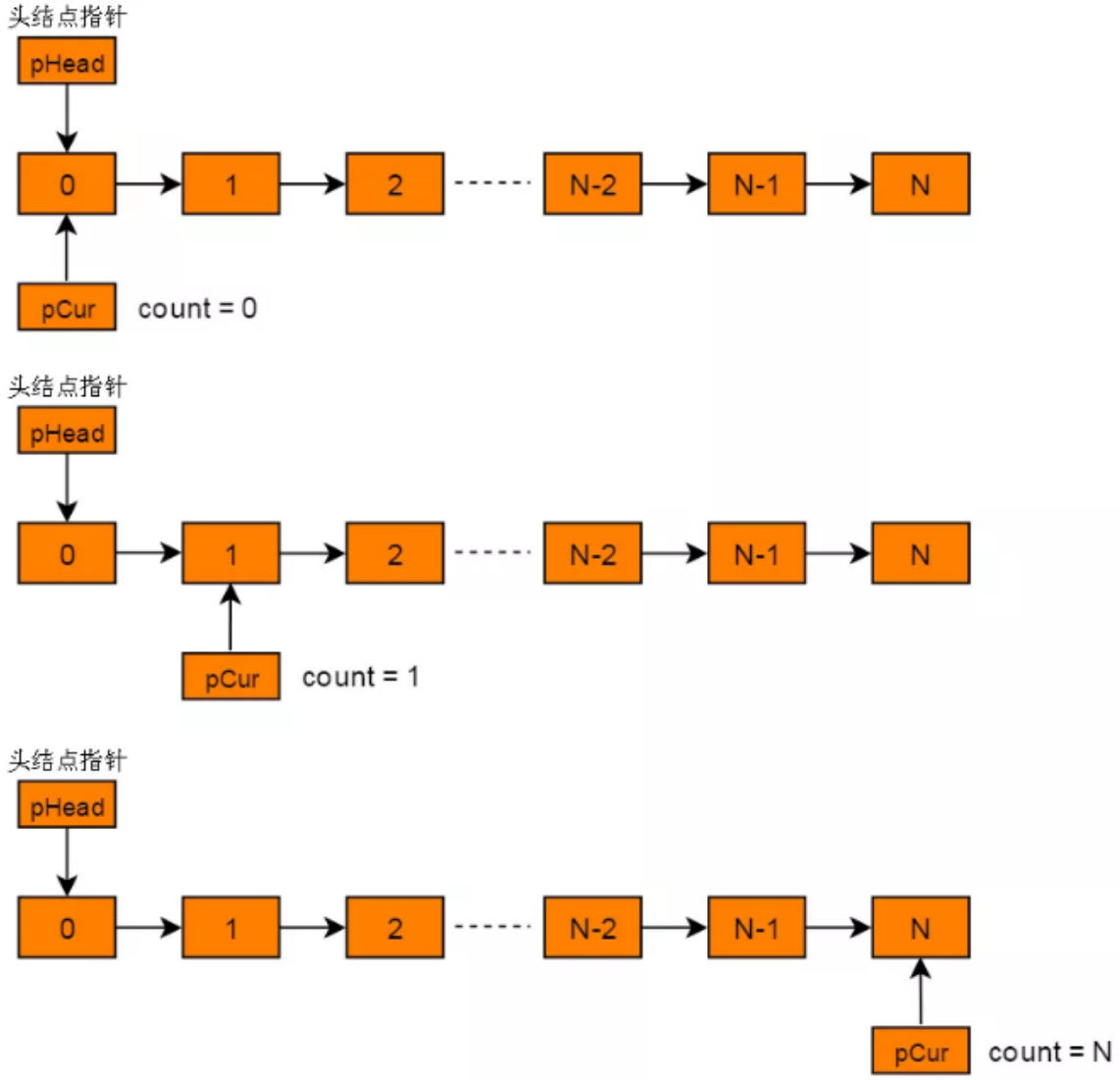


图 1

1.2.3 代码实现

```
/*计算链表长度*/
int listLength(ListNode* pHead){
    int count = 0;
```

```
    ListNode* pCur = pHead->next;
    if(pCur == NULL){
        printf("error");
    }
    while(pCur){
        count++;
         pCur = pCur->pNext;
     }
     return count;
}
/*查找第k个节点的值*/
ListNode* searchNodeK(ListNode* pHead, int k){
    int i = 0;
    ListNode* pCur = pHead;
    //计算链表长度
    int len = listLength(pHead);
    if(k > len){
        printf("error");
    }
    //循环len-k+1次
    for(i=0; i < len-k+1; i++){
        pCur  = pCur->next;
    }
    return pCur;//返回倒数第K个节点
}
```

采用这种遍历方式需要两次遍历链表，时间复杂度为O(n※2)。可见这种方式最为简单,也较好理解，但是效率低下。

1.3 双指针法

1.3.1 解题思想

（1）定义两个指针 p1 和 p2 分别指向链表头节点。

（2）p1 前进 K 个节点，则 p1 与 p2 相距 K 个节点。

（3）p1，p2 同时前进，每次前进 1 个节点。

（4）当 p1 指向到达链表末尾，由于 p1 与 p2 相距 K 个节点，则 p2 指向目标节点。

1.3.2 图解过程

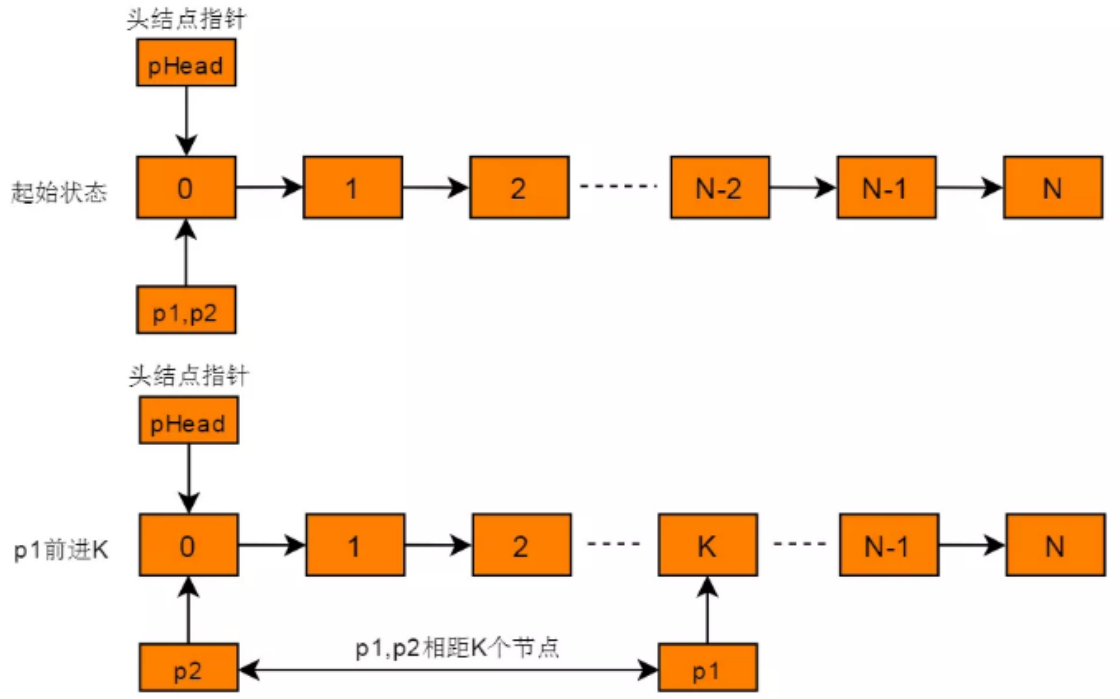
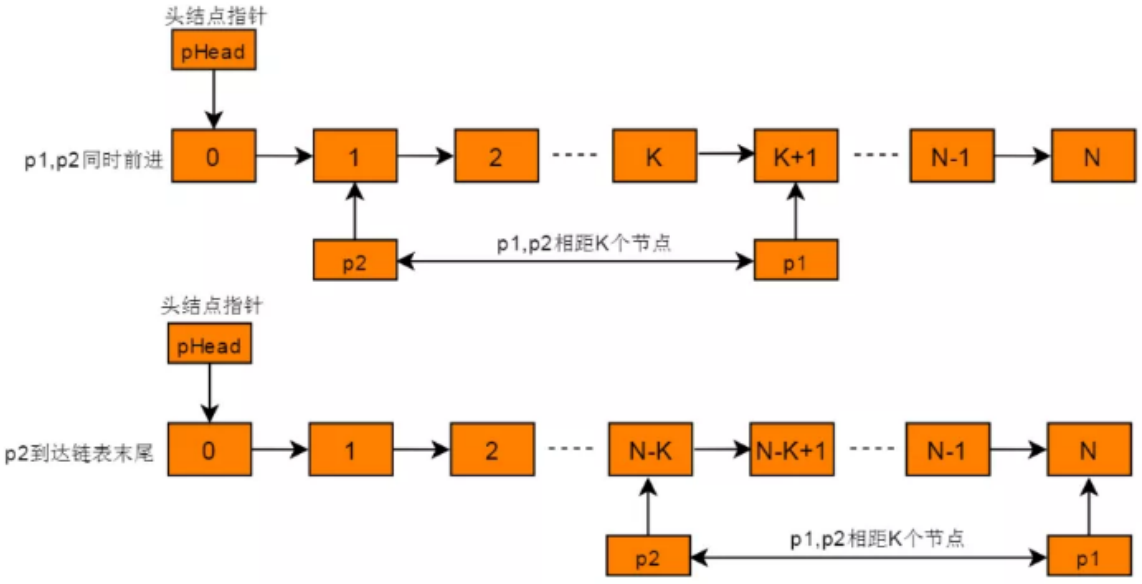


图 3



1.3.3 代码实现

```
ListNode* findKthTail(ListNode *pHead, int K){
    if (NULL == pHead || K == 0)
        return NULL;
    //p1，p2均指向头节点
    ListNode *p1 = pHead;
    ListNode *p2 = pHead;
    //p1先出发，前进K个节点
    for (int i = 0; i < K; i++) {
        if (p1)//防止k大于链表节点的个数
            p1 = p1->_next;
         else
```

```
            return NULL;
    }

    while (p1)//如果p1没有到达链表结尾，则p1，p2继续遍历
    {
        p1 = p1->_next;
        p2 = p2->_next;
    }
    return p2;//当p1到达末尾时，p2正好指向倒数第K个节点
}
```

可以看出使用双指针法只需遍历链表一次，这种方法更为高效时间复杂度为O(n)，通常笔试题目中要考的也是这种方法。

3 链表中存在环问题

3.1 判断链表是否有环

单链表中的环是指链表末尾的节点的 next 指针不为 NULL ，而是指向了链表中的某个节点，导致链表中出现了环形结构。

链表中有环示意图：

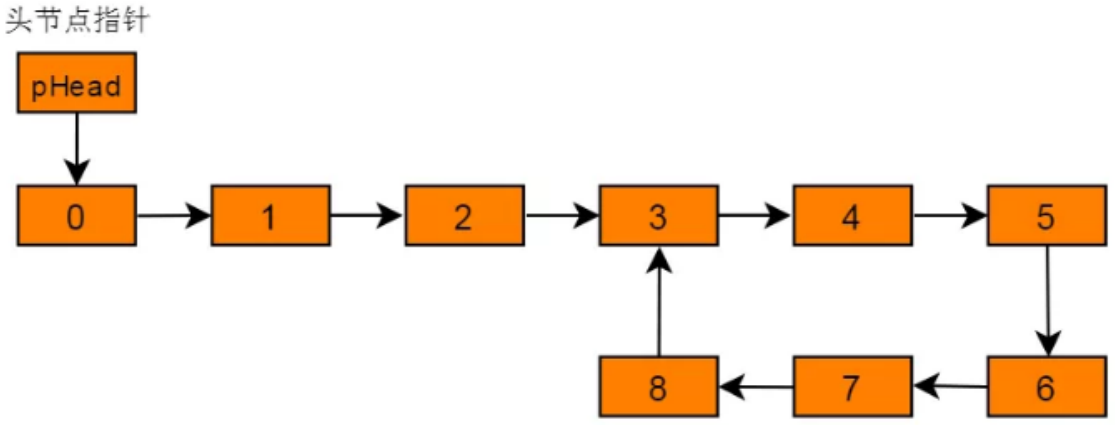


图 5

3.1.3 快慢指针法

3.1.3.1 解题思想

（1）定义两个指针分别为 slow，fast，并且将指针均指向链表头节点。

（2）规定，slow 指针每次前进 1 个节点，fast 指针每次前进两个节点。

（3）当 slow 与 fast 相等，且二者均不为空，则链表存在环。

3.1.3.3 代码实现

```
bool isExistLoop(ListNode* pHead)  {
    ListNode* fast;//慢指针，每次前进一个节点
```

```
    ListNode* slow;//快指针，每次前进2个节点
    slow = fast = pHead ;  //两个指针均指向链表头节点
    //当没有到达链表结尾，则继续前进
    while (slow != NULL && fast -> next != NULL)  {
        slow = slow -> next ; //慢指针前进一个节点
        fast = fast -> next -> next ; //快指针前进两个节点
        if (slow == fast)  //若两个指针相遇，且均不为NULL则存在环
            return true ;
     }
     //到达末尾仍然没有相遇，则不存在环
     return false ;
}
```

3.2 定位环入口

在 3.1 节中，已经实现了链表中是否有环的判断方法。那么，当链表中存在环，如何确定环的入口节点呢?

slow 指针每次前进一个节点，故 slow 与 fast 相遇时，slow 还没有遍历完整个链表。设 slow 走过节点数为 s，fast 走过节点数为 2s。设环入口点距离头节点为 a，slow 与 fast 首次相遇点距离入口点为 b，环的长度为 r。

则有:

s = a + b;

2s = n * r + a + b; n 代表fast指针已经在环中循环的圈数。

则推出:

s = n * r; 意味着slow指针走过的长度为环的长度整数倍。

若链表头节点到环的末尾节点度为 L，slow 与 fast 的相遇节点距离环入口节点为 X。

则有:

a+X = s = n * r = (n - 1) * r + (L - a);

a = (n - 1) * r + (L - a - X);

上述等式可以看出:

从 slow 与 fast 相遇点出发一个指针 p1，请进 (L - a - X) 步，则此指针到达入口节点。同时指针 p2 从头结点出发，前进 a 步。当 p1 与 p2 相遇时，此时 p1 与 p2 均指向入口节点。

例如图3.1所示链表:

slow 走过节点 s = 6;

fast 走过节点 2s = 12;

环入口节点据流头节点 a = 3;

相遇点距离头节点 X = 3;

L = 8;

r = 6;

可以得出 a = (n - 1) * r + (L - a - X)结果成立。

3.2.3 代码实现

```
//找到环中的相遇节点
ListNode* getMeetingNode(ListNode* pHead) // 假设为带头节点的单链表
{
    ListNode* fast;//慢指针，每次前进一个节点
    ListNode* slow;//快指针，每次前进2个节点
    slow = fast = pHead ;  //两个指针均指向链表头节点
    //当没有到达链表结尾，则继续前进
    while (slow != NULL && fast -> next != NULL){
        slow = slow -> next ; //慢指针前进一个节点
         fast = fast -> next -> next ; //快指针前进两个节点
         if (slow == fast)  //若两个指针相遇，且均不为NULL则存在环
             return slow;
     }

    //到达末尾仍然没有相遇，则不存在环
    return NULL ;
}
//找出环的入口节点
ListNode* getEntryNodeOfLoop(ListNode* pHead){
    ListNode* meetingNode = getMeetingNode(pHead); // 先找出环中的相遇节点
    if (meetingNode == NULL)
        return NULL;
    ListNode* p1 = meetingNode;
    ListNode* p2 = pHead;
    while (p1 != p2) // p1和p2以相同的速度向前移动，当p2指向环的入口节点时，p1已经围绕着环走了n圈又回到了入口节点。
    {
        p1 = p1->next;
        p2 = p2->next;
    }
    //返回入口节点
    return p1;
}
```

3.3 计算环长度

在3.1中找到了 slow 与 fast 的相遇节点，令 solw 与 fast 指针从相遇节点出发，按照之前的前进规则，当 slow 与fast 再次相遇时，slow 走过的长度正好为环的长度。

3.3.3 代码实现

```
int getLoopLength(ListNode* head){
    ListNode* slow = head;
    ListNode* fast = head;
    while ( fast && fast->next ){
        slow = slow->next;
        fast = fast->next->next;
        if ( slow == fast )//第一次相遇
            break;
    }
     //slow与fast继续前进
     slow = slow->next;
     fast = fast->next->next;
     int length = 1;        //环长度
     while ( fast != slow )//再次相遇
     {
         slow = slow->next;
         fast = fast->next->next;
         length ++;         //累加
     }
     //当slow与fast再次相遇，得到环长度
     return length;
}
```

4 使用链表实现大数加法

4.1 问题描述

两个用链表代表的整数，其中每个节点包含一个数字。数字存储按照在原来整数中相反的顺序，使得第一个数字位于链表的开头。写出一个函数将两个整数相加，用链表形式返回和。

```
例如：
输入：
3->1->5->null
5->9->2->null,
输出：
8->0->8->null
```

4.2 代码实现

```
ListNode* numberAddAsList(ListNode* l1, ListNode* l2) {
        ListNode *ret = l1, *pre = l1;
        int up = 0;
        while (l1 != NULL && l2 != NULL) {
            //数值相加
            l1->val = l1->val + l2->val + up;
            //计算是否有进位
            up = l1->val / 10;
            //保留计算结果的个位
             l1->val %= 10;
             //记录当前节点位置
             pre = l1;
             //同时向后移位
             l1 = l1->next;
             l2 = l2->next;
         }
         //若l1到达末尾，说明l1长度小于l2
         if (l1 == NULL)
             //pre->next指向l2的当前位置
             pre->next = l2;
         //l1指针指向l2节点当前位置
         l1 = pre->next;
         //继续计算剩余节点
         while (l1 != NULL) {
             l1->val = l1->val + up;
             up = l1->val / 10;
             l1->val %= 10;
             pre = l1;
             l1 = l1->next;
         }

         //最高位计算有进位，则新建一个节点保留最高位
         if (up != 0) {
             ListNode *tmp = new ListNode(up);
             pre->next = tmp;
         }
         //返回计算结果链表
         return ret;
}
```

5 有序链表合并

5.1 问题描述

题目：将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

> 示例：
> 输入：
> 1->2->4,
> 1->3->4
> 输出：
> 1->1->2->3->4->4

5.2.1 解题思想

（1）对空链表存在的情况进行处理，假如 pHead1 为空则返回 pHead2 ， pHead2 为空则返回 pHead1。（两个都为空此情况在pHead1为空已经被拦截）

（2）在两个链表无空链表的情况下确定第一个结点，比较链表1和链表2的第一个结点的值，将值小的结点保存下来为合并后的第一个结点。并且把第一个结点为最小的链表向后移动一个元素。

（3）继续在剩下的元素中选择小的值，连接到第一个结点后面，并不断next将值小的结点连接到第一个结点后面，直到某一个链表为空。

（4）当两个链表长度不一致时，也就是比较完成后其中一个链表为空，此时需要把另外一个链表剩下的元素都连接到第一个结点的后面。

5.2.2 代码实现

```
Node* Merge(Node* head1, Node* head2) //合并两个有序链表成一个有序链表
{
 if(head1 == NULL)
 return head2;
 if(head2 == NULL)
 return head1;
 Node *head, *p1, *p2;
 if(head1->data < head2->data) //确定好合并后的头结点
 {
  head = head1;
  p1 = head1->next;
  p2 = head2;
  }
  else
  {
  head = head2;
```

```
17  p1 = head1;
18  p2 = head2->next;
19  }
20  Node *move = head; //定义一个移动指针,将两个链表连接在一起
21  while(p1 && p2)
22  {
23  if(p1->data <= p2->data)
24  {
25  move->next = p1;
26  move = p1;
27  p1 = p1->next;
28  }
29  else if(p1->data > p2->data)
30  {
31  move->next = p2;
32  move = p2;
33  p2 = p2->next;
34  }
35  }
36  if(p1)
37  move->next = p1; // 补齐p1剩余未比较的节点
38  if(p2)
39  move->next = p2; // 补齐p2剩余未比较的节点
40  return head;
41 }
42
```

6 删除链表中节点，要求时间复杂度为O(1)

6.1 问题描述

给定一个单链表中的表头和一个等待被删除的节点。请在 O(1) 时间复杂度删除该链表节点。并在删除该节点后，返回表头。

示例：

给定 1->2->3->4，和节点 3，返回 1->2->4。

题目描述：给定链表的头指针和一个节点指针，在O(1)时间删除该节点。[Google 面试题]

分析：本题与《编程之美》上的「从无头单链表中删除节点」类似。主要思想都是「狸猫换太子」，即用下一个节点数据覆盖要删除的节点，然后删除下一个节点。但是如果节点是尾节点时，该方法就行不通了。

代码：如下

代码：如下

```
//O(1)时间删除链表节点，从无头单链表中删除节点
void deleteRandomNode(Node *cur)
{
    assert(cur != NULL);
    assert(cur->next != NULL);    //不能是尾节点
    Node* pNext = cur->next;
    cur->data = pNext->data;
    cur->next = pNext->next;
    delete pNext;
}
```

6.2 解题思想

在之前介绍的单链表删除节点中，最普通的方法就是遍历链表，复杂度为O(n)。

如果我们把删除节点的下一个结点的值赋值给要删除的结点，然后删除这个结点，这相当于删除了需要删除的那个结点。因为我们很容易获取到删除节点的下一个节点，所以复杂度只需要O(1)。

> 示例
> 单链表：1->2->3->4->NULL
> 若要删除节点 3 。第一步将节点3的下一个节点的值4赋值给当前节点。变成 1->2->4->4->NULL，然后将就 4 这个结点删除，就达到目的了。 1->2->4->NULL

如果删除的节点的是头节点，把头结点指向 NULL。

如果删除的节点的是尾节点，那只能从头遍历到头节点的上一个结点。

6.3 代码实现

```
1  void deleteNode(ListNode **pHead, ListNode* pDelNode) {
2          if(pDelNode == NULL)
3              return;
4          if(pDelNode->next != NULL){
5              ListNode *pNext = pDelNode->next;
6              //下一个节点值赋给待删除节点
7              pDelNode->val   =  pNext->val;
8              //待删除节点指针指后面第二个节点
9              pDelNode->next  = pNext->next;
```

```
            //删除待删除节点的下一个节点
            delete pNext;
            pNext = NULL;
        }else if(*pHead == pDelNode)//删除的节点是头节点
         {
            delete pDelNode;
            pDelNode= NULL;
            *pHead = NULL;
        } else//删除的是尾节点
        {
            ListNode *pNode = *pHead;
            while(pNode->next != pDelNode) {
                pNode = pNode->next;
            }
            pNode->next = NULL;
            delete pDelNode;
            pDelNode= NULL;
        }
    }
```

7 反转链表

反转一个单链表。

示例:

输入： 1->2->3->4->5->NULL

输出： 5->4->3->2->1->NULL

进阶:

你可以迭代或递归地反转链表。你能否用两种方法解决这道题?

8.2 解题思路

设置三个节点pre、cur、next

- （1）每次查看cur节点是否为NULL，如果是，则结束循环，获得结果
- （2）如果cur节点不是为NULL，则先设置临时变量next为cur的下一个节点
- （3）让cur的下一个节点变成指向pre，而后pre移动cur，cur移动到next
- （4）重复（1）（2）（3）

8.3 代码实现

```
ListNode* reverseList(ListNode* head) {
        ListNode* pre = NULL;
        ListNode* cur = head;
        while(cur != NULL){
```

```
            ListNode* next = cur->next;
            cur->next = pre;
            pre = cur;
            cur = next;
        }
        return pre;
    }
```

9 求链表中间节点

题目描述：求链表的中间节点，如果链表的长度为偶数，返回中间两个节点的任意一个，若为奇数，则返回中间节点。

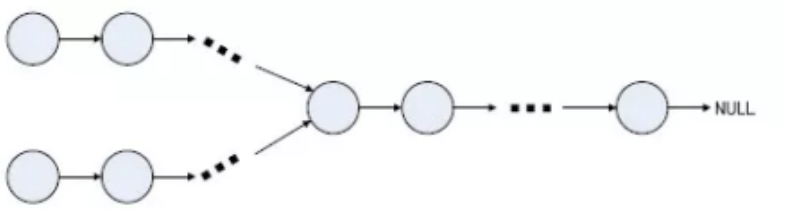
分析：此题的解决思路和第3题「求链表的倒数第 k 个节点」很相似。可以先求链表的长度，然后计算出中间节点所在链表顺序的位置。但是如果要求只能扫描一遍链表，如何解决呢？最高效的解法和第3题一样，通过两个指针来完成。用两个指针从链表头节点开始，一个指针每次向后移动两步，一个每次移动一步，直到快指针移到到尾节点，那么慢指针即是所求。

): 如下

```
//求链表的中间节点
Node* theMiddleNode(Node *head)
{
    if(head == NULL)
        return NULL;
    Node *slow,*fast;
    slow = fast = head;
    //如果要求在链表长度为偶数的情况下，返回中间两个节点的第一个，可以用下面的循环条件
    //while(fast && fast->next != NULL && fast->next->next != NULL)
    while(fast != NULL && fast->next != NULL)
    {
        fast = fast->next->next;
        slow = slow->next;
    }
    return slow;
}
```

10 判断两个链表是否相交

题目描述：给出两个单向链表的头指针（如下图所示），判断这两个链表是否相交。这里为了简化问题，我们假设两个链表均不带环。



分析：

“如果两个没有环的链表相交于某一节点，那么在这个节点之后的所有节点都是两个链表共有的”这个特点，我们可以知道，如果它们相交，则最后一个节点一定是共有的。而我们很容易能得到链表的最后一个节点，所以这成了我们简化解法的一个主要突破口。那么，我们只要判断两个链表的尾指针是否相等。相等，则链表相交；否则，链表不相交。

所以，先遍历第一个链表，记住最后一个节点。然后遍历第二个链表，到最后一个节点时和第一个链表的最后一个节点做比较，如果相同，则相交，否则，不相交。这样我们就得到了一个时间复杂度，它为O((Length(h1) + Length(h2))，而且只用了一个额外的指针来存储最后一个节点。这个方法时间复杂度为线性O(N)，空间复杂度为O(1)，显然比解法三更胜一筹。

：如下

```
//判断两个链表是否相交
bool isIntersect(Node *h1,Node *h2)
{
    if(h1 == NULL || h2 == NULL) return false;    //异常判断
    while(h1->next != NULL)
    {
        h1 = h1->next;
    }

    while(h2->next != NULL)
    {
        h2 = h2->next;
    }

    if(h1 == h2) return true;        //尾节点是否相同
    else return false;
}
```

11. 两个链表相交的第一个公共节点

题目描述：如果两个无环单链表相交，怎么求出他们相交的第一个节点呢？

分析：采用对齐的思想。计算两个链表的长度 L1，L2，分别用两个指针 p1，p2 指向两个链表的头，然后将较长链表的 p1（假设为 p1）向后移动L2 - L1个节点，然后再同时向后移动p1，p2，直到 p1 = p2。相遇的点就是相交的第一个节点。

代码：如下

码：如下

```
//求两链表相交的第一个公共节点
Node* findIntersectNode(Node *h1,Node *h2)
{
    int len1 = listLength(h1);          //求链表长度
    int len2 = listLength(h2);
    //对齐两个链表
    if(len1 > len2)
    {
        for(int i=0;i<len1-len2;i++)
            h1=h1->next;
    }
    else
    {
        for(int i=0;i<len2-len1;i++)
            h2=h2->next;
    }

    while(h1 != NULL)
    {
        if(h1 == h2)
            return h1;
        h1 = h1->next;
        h2 = h2->next;
    }
    return NULL;
}
```