# 1、链表

**程序员面试题精选**(09)－查找链表中倒数第k个结点

题目：输入一个单向链表，输出该链表中倒数第k个结点。链表的倒数第0个结点为链表的尾指针。链表结点定义如下： struct ListNode

{

int m\_nKey;

ListNode\* m\_pNext;

};

分析：为了得到倒数第k个结点，很自然的想法是先走到链表的尾端，再从尾端回溯k步。可是输入的是单向链表，只有从前往后的指针而没有从后往前的指针。因此我们需要打开我们的思路。

既然不能从尾结点开始遍历这个链表，我们还是把思路回到头结点上来。假设整个链表有n个结点，那么倒数第k个结点是从头结点开始的第n-k-1个结点（从0开始计数）。如果我们能够得到链表中结点的个数n，那我们只要从头结点开始往后走n-k-1步就可以了。如何得到结点数n？这个不难，只需要从头开始遍历链表，每经过一个结点，计数器加一就行了。

这种思路的时间复杂度是O(n)，但需要遍历链表两次。第一次得到链表中结点个数n，第二次得到从头结点开始的第n?-k-1个结点即倒数第k个结点。

如果链表的结点数不多，这是一种很好的方法。但如果输入的链表的结点个数很多，有可能不能一次性把整个链表都从硬盘读入物理内存，那么遍历两遍意味着一个结点需要两次从硬盘读入到物理内存。我们知道把数据从硬盘读入到内存是非常耗时间的操作。我们能不能把链表遍历的次数减少到1？如果可以，将能有效地提高代码执行的时间效率。

如果我们在遍历时维持两个指针，第一个指针从链表的头指针开始遍历，在第k-1步之前，第二个指针保持不动；在第k-1步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在k-1，当第一个（走在前面的）指针到达链表的尾结点时，第二个指针（走在后面的）指针正好是倒数第k个结点。

这种思路只需要遍历链表一次。对于很长的链表，只需要把每个结点从硬盘导入到内存一次。因此这一方法的时间效率前面的方法要高。

思路一的参考代码：

///////////////////////////////////////////////////////////////////////

// Find the kth node from the tail of a list

// Input: pListHead - the head of list

// k - the distance to the tail

// Output: the kth node from the tail of a list

///////////////////////////////////////////////////////////////////////

ListNode\* FindKthToTail\_Solution1(ListNode\* pListHead, unsigned int k)

{

if(pListHead == NULL)

return NULL;

// count the nodes number in the list

ListNode \*pCur = pListHead;

unsigned int nNum = 0;

while(pCur->m\_pNext != NULL)

{

pCur = pCur->m\_pNext;

nNum ++;

}

// if the number of nodes in the list is less than k

// do nothing

if(nNum < k)

return NULL;

// the kth node from the tail of a list

// is the (n - k)th node from the head

pCur = pListHead;

for(unsigned int i = 0; i < nNum - k; ++ i)

pCur = pCur->m\_pNext;

return pCur;

}

思路二的参考代码：

///////////////////////////////////////////////////////////////////////

// Find the kth node from the tail of a list

// Input: pListHead - the head of list

// k - the distance to the tail

// Output: the kth node from the tail of a list

///////////////////////////////////////////////////////////////////////

ListNode\* FindKthToTail\_Solution2(ListNode\* pListHead, unsigned int k)

{

if(pListHead == NULL)

return NULL;

ListNode \*pAhead = pListHead;

ListNode \*pBehind = NULL;

for(unsigned int i = 0; i < k; ++ i)

{

if(pAhead->m\_pNext != NULL)

pAhead = pAhead->m\_pNext;

else

{

// if the number of nodes in the list is less than k,

// do nothing

return NULL;

}

}

pBehind = pListHead;

// the distance between pAhead and pBehind is k

// when pAhead arrives at the tail, p

// Behind is at the kth node from the tail

while(pAhead->m\_pNext != NULL)

{

pAhead = pAhead->m\_pNext;

pBehind = pBehind->m\_pNext;

}

return pBehind;

}

讨论：这道题的代码有大量的指针操作。在软件开发中，错误的指针操作是大部分问题的根源。因此每个公司都希望程序员在操作指针时有良好的习惯，比如使用指针之前判断是不是空指针。这些都是编程的细节，但如果这些细节把握得不好，很有可能就会和心仪的公司失之交臂。

另外，这两种思路对应的代码都含有循环。含有循环的代码经常出的问题是在循环结束条件的判断。是该用小于还是小于等于？是该用k还是该用k-1？由于题目要求的是从0开始计数，而我们的习惯思维是从1开始计数，因此首先要想好这些边界条件再开始编写代码，再者要在编写完代码之后再用边界值、边界值减1、边界值加1都运行一次（在纸上写代码就只能在心里运行了）。

扩展：和这道题类似的题目还有：输入一个单向链表。如果该链表的结点数为奇数，输出中间的结点；如果链表结点数为偶数，输出中间两个结点前面的一个。如果各位感兴趣，请自己分析并编写代码。

**程序员面试题精选**(19)－反转链表

题目：输入一个链表的头结点，反转该链表，并返回反转后链表的头结点。链表结点定义如下：

struct ListNode

{

int m\_nKey;

ListNode\* m\_pNext;

};

分析：这是一道广为流传的微软面试题。由于这道题能够很好的反应出程序员思维是否严密，在微软之后已经有很多公司在面试时采用了这道题。

为了正确地反转一个链表，需要调整指针的指向。与指针操作相关代码总是容易出错的，因此最好在动手写程序之前作全面的分析。在面试的时候不急于动手而是一开始做仔细的分析和设计，将会给面试官留下很好的印象，因为在实际的软件开发中，设计的时间总是比写代码的时间长。与其很快地写出一段漏洞百出的代码，远不如用较多的时间写出一段健壮的代码。

为了将调整指针这个复杂的过程分析清楚，我们可以借助图形来直观地分析。假设下图中l、m和n是三个相邻的结点：

a?b?…?l m?n?…

假设经过若干操作，我们已经把结点l之前的指针调整完毕，这些结点的m\_pNext指针都指向前面一个结点。现在我们遍历到结点m。当然，我们需要把调整结点的m\_pNext指针让它指向结点l。但注意一旦调整了指针的指向，链表就断开了，如下图所示：

a?b?…l?m n?…

因为已经没有指针指向结点n，我们没有办法再遍历到结点n了。因此为了避免链表断开，我们需要在调整m的m\_pNext之前要把n保存下来。

接下来我们试着找到反转后链表的头结点。不难分析出反转后链表的头结点是原始链表的尾位结点。什么结点是尾结点？就是m\_pNext为空指针的结点。

基于上述分析，我们不难写出如下代码：

///////////////////////////////////////////////////////////////////////

// Reverse a list iteratively

// Input: pHead - the head of the original list

// Output: the head of the reversed head

///////////////////////////////////////////////////////////////////////

ListNode\* ReverseIteratively(ListNode\* pHead)

{

ListNode\* pReversedHead = NULL;

ListNode\* pNode = pHead;

ListNode\* pPrev = NULL;

while(pNode != NULL)

{

// get the next node, and save it at pNext

ListNode\* pNext = pNode->m\_pNext;

// if the next node is null, the currect is the end of original

// list, and it's the head of the reversed list

if(pNext == NULL)

pReversedHead = pNode;

// reverse the linkage between nodes

pNode->m\_pNext = pPrev;

// move forward on the the list

pPrev = pNode;

pNode = pNext;

}

return pReversedHead;

}

扩展：本题也可以递归实现。感兴趣的读者请自己编写递归代码。

**程序员面试题精选**100题(33)-在O(1)时间删除链表结点

题目：给定链表的头指针和一个结点指针，在O(1) 时间删除该结点。链表结点的定义如下：

struct ListNode

{

int m\_nKey;

ListNode\* m\_pNext;

};

函数的声明如下：

void DeleteNode(ListNode\* pListHead, ListNode\* pToBeDeleted);

分析：这是一道广为流传的Google 面试题，能有效考察我们的编程基本功，还能考察我们的反应速度，更重要的是，还能考察我们对时间复杂度的理解。

在链表中删除一个结点，最常规的做法是从链表的头结点开始，顺序查找要删除的结点，找到之后再删除。由于需要顺序查找，时间复杂度自然就是O(n) 了。

我们之所以需要从头结点开始查找要删除的结点，是因为我们需要得到要删除的结点的前面一个结点。我们试着换一种思路。我们可以从给定的结点得到它的下一个 结点。这个时候我们实际删除的是它的下一个结点，由于我们已经得到实际删除的结点的前面一个结点，因此完全是可以实现的。当然，在删除之前，我们需要需要 把给定的结点的下一个结点的数据拷贝到给定的结点中。此时，时间复杂度为O(1) 。

上面的思路还有一个问题：如果删除的结点位于链表的尾部，没有下一个结点，怎么办？我们仍然从链表的头结点开始，顺便遍历得到给定结点的前序结点，并完成删除操作。这个时候时间复杂度是O(n) 。

那题目要求我们需要在O(1) 时间完成删除操作，我们的算法是不是不符合要求？实际上，假设链表总共有n 个结点，我们的算法在n-1 总情况下时间复杂度是O(1) ，只有当给定的结点处于链表末尾的时候，时间复杂度为O(n) 。那么平均时间复杂度[(n-1)\*O(1)+O(n)]/n ，仍然为O(1) 。

基于前面的分析，我们不难写出下面的代码。

参考代码：

///////////////////////////////////////////////////////////////////////

// Delete a node in a list

// Input: pListHead - the head of list

// pToBeDeleted - the node to be deleted

///////////////////////////////////////////////////////////////////////

void DeleteNode(ListNode\* pListHead, ListNode\* pToBeDeleted)

{

if(!pListHead || !pToBeDeleted)

return;

// if pToBeDeleted is not the last node in the list

if(pToBeDeleted->m\_pNext != NULL)

{

// copy data from the node next to pToBeDeleted

ListNode\* pNext = pToBeDeleted->m\_pNext;

pToBeDeleted->m\_nKey = pNext->m\_nKey;

pToBeDeleted->m\_pNext = pNext->m\_pNext;

// delete the node next to the pToBeDeleted

delete pNext;

pNext = NULL;

}

// if pToBeDeleted is the last node in the list

else

{

// get the node prior to pToBeDeleted

ListNode\* pNode = pListHead;

while(pNode->m\_pNext != pToBeDeleted)

{

pNode = pNode->m\_pNext;

}

// deleted pToBeDeleted

pNode->m\_pNext = NULL;

delete pToBeDeleted;

pToBeDeleted = NULL;

}

}

值得注意的是，为了让代码看起来简洁一些，上面的代码基于两个假设：（1 ）给定的结点的确在链表中；（2 ）给定的要删除的结点不是链表的头结点。不考虑第一个假设对代码的鲁棒性是有影响的。至于第二个假设，当整个列表只有一个结点时，代码会有问题。但这个假 设不算很过分，因为在有些链表的实现中，会创建一个虚拟的链表头，并不是一个实际的链表结点。这样要删除的结点就不可能是链表的头结点了。当然，在面试 中，我们可以把这些假设和面试官交流。这样，面试官还是会觉得我们考虑问题很周到的。

**程序员面试题精选**100题(35)-找两个链表的第一个公共结点

题目：两个单向链表，找出它们的第一个公共结点。

链表的结点定义为：

struct ListNode

{

int m\_nKey;

ListNode\* m\_pNext;

};

分析：这是一道微软的面试题。微软非常喜欢与链表相关的题目，因此在微软的面试题中，链表出现的概率相当高。

如果两个单向链表有公共的结点，也就是说两个链表从某一结点开始，它们的m\_pNext 都指向同一个结点。但由于是单向链表的结点，每个结点只有一个m\_pNext ，因此从第一个公共结点开始，之后它们所有结点都是重合的，不可能再出现分叉。所以，两个有公共结点而部分重合的链表，拓扑形状看起来像一个Y ，而不可能像X 。

看到这个题目，第一反应就是蛮力法：在第一链表上顺序遍历每个结点。每遍历一个结点的时候，在第二个链表上顺序遍历每个结点。如果此时两个链表上的结点是一样的，说明此时两个链表重合，于是找到了它们的公共结点。如果第一个链表的长度为m ，第二个链表的长度为n ，显然，该方法的时间复杂度为O(mn) 。

接下来我们试着去寻找一个线性时间复杂度的算法。我们先把问题简化：如何判断两个单向链表有没有公共结点？前面已经提到， 如果两个链表有一个公共结点，那么该公共结点之后的所有结点都是重合的。那么，它们的最后一个结点必然是重合的。因此，我们判断两个链表是不是有重合的部 分，只要分别遍历两个链表到最后一个结点。如果两个尾结点是一样的，说明它们用重合；否则两个链表没有公共的结点。

在上面的思路中，顺序遍历两个链表到尾结点的时候，我们不能保证在两个链表上同时到达尾结点。这是因为两个链表不一定长度一样。但如果假设一个链表比另一个长l 个结点，我们先在长的链表上遍历l 个结点，之后再同步遍历，这个时候我们就能保证同时到达最后一个结点了。由于两个链表从第一个公共结点考试到链表的尾结点，这一部分是重合的。因此，它们肯定也是同时到达第一公共结点的。于是在遍历中，第一个相同的结点就是第一个公共的结点。

在这个思路中，我们先要分别遍历两个链表得到它们的长度，并求出两个长度之差。在长的链表上先遍历若干次之后，再同步遍历两个链表，知道找到相同的结点，或者一直到链表结束。此时，如果第一个链表的长度为m ，第二个链表的长度为n ，该方法的时间复杂度为O(m+n) 。

基于这个思路，我们不难写出如下的代码：

///////////////////////////////////////////////////////////////////////

// Find the first common node in the list with head pHead1 and

// the list with head pHead2

// Input: pHead1 - the head of the first list

// pHead2 - the head of the second list

// Return: the first common node in two list. If there is no common

// nodes, return NULL

///////////////////////////////////////////////////////////////////////

ListNode \* FindFirstCommonNode( ListNode \*pHead1, ListNode \*pHead2)

{

// Get the length of two lists

unsigned int nLength1 = ListLength(pHead1);

unsigned int nLength2 = ListLength(pHead2);

int nLengthDif = nLength1 - nLength2;

// Get the longer list

ListNode \*pListHeadLong = pHead1;

ListNode \*pListHeadShort = pHead2;

if(nLength2 > nLength1)

{

pListHeadLong = pHead2;

pListHeadShort = pHead1;

nLengthDif = nLength2 - nLength1;

}

// Move on the longer list

for(int i = 0; i < nLengthDif; ++ i)

pListHeadLong = pListHeadLong->m\_pNext;

// Move on both lists

while((pListHeadLong != NULL) &&

(pListHeadShort != NULL) &&

(pListHeadLong != pListHeadShort))

{

pListHeadLong = pListHeadLong->m\_pNext;

pListHeadShort = pListHeadShort->m\_pNext;

}

// Get the first common node in two lists

ListNode \*pFisrtCommonNode = NULL;

if(pListHeadLong == pListHeadShort)

pFisrtCommonNode = pListHeadLong;

return pFisrtCommonNode;

}

///////////////////////////////////////////////////////////////////////

// Get the length of list with head pHead

// Input: pHead - the head of list

// Return: the length of list

///////////////////////////////////////////////////////////////////////

unsigned int ListLength(ListNode\* pHead)

{

unsigned int nLength = 0;

ListNode\* pNode = pHead;

while(pNode != NULL)

{

++ nLength;

pNode = pNode->m\_pNext;

}

return nLength;

}

**程序员面试题精选**（35）：一次遍历链表求中间节点位置

思路：声明两指针p和q，p每往后移动两节点，q往后移动一个节点，当p->next==NULL时，q便是中间节点的位置。

知道思路后，实现就比较简单了，在此不给出代码。

**程序员面试题精选**（53）：删除链表结点（时间复杂度为O（1））

题目：给定链表的头指针和一个结点指针，在O(1)时间删除该结点。链表结点的定义如下：

struct ListNode

{

int m\_nKey;

ListNode\* m\_pNext;

};

函数的声明如下：

void DeleteNode(ListNode\* pListHead, ListNode\* pToBeDeleted);

分析：这是一道广为流传的Google面试题，能有效考察我们的编程基本功，还能考察我们的反应速度，更重要的是，还能考察我们对时间复杂度的理解。

在链表中删除一个结点，最常规的做法是从链表的头结点开始，顺序查找要删除的结点，找到之后再删除。由于需要顺序查找，时间复杂度自然就是O(n) 了。

我们之所以需要从头结点开始查找要删除的结点， 是因为我们需要得到要删除的结点的前面一个结点。我们试着换一种思路。我们可以从给定的结点得到它的下一个结点。这个时候我们实际删除的是它的下一个结 点，由于我们已经得到实际删除的结点的前面一个结点，因此完全是可以实现的。当然，在删除之前，我们需要需要把给定的结点的下一个结点的数据拷贝到给定的 结点中。此时，时间复杂度为O(1)。

上面的思路还有一个问题：如果删除的结点位于链表的尾部，没有下一个结点，怎么办？我们仍然从链表的头结点开始，顺便遍历得到给定结点的前序结点，并完成删除操作。这个时候时间复杂度是O(n)。

那题目要求我们需要在O(1)时间完成删除操 作，我们的算法是不是不符合要求？实际上，假设链表总共有n个结点，我们的算法在n-1总情况下时间复杂度是O(1)，只有当给定的结点处于链表末尾的时 候，时间复杂度为O(n)。那么平均时间复杂度[(n-1)\*O(1)+O(n)]/n，仍然为O(1)。

基于前面的分析，我们不难写出下面的代码。

参考代码：

///////////////////////////////////////////////////////////////////////

// Delete a node in a list

// Input: pListHead - the head of list

// pToBeDeleted - the node to be deleted

///////////////////////////////////////////////////////////////////////

void DeleteNode(ListNode\* pListHead, ListNode\* pToBeDeleted)

{

if(!pListHead || !pToBeDeleted)

return;

// if pToBeDeleted is not the last node in the list

if(pToBeDeleted->m\_pNext != NULL)

{

// copy data from the node next to pToBeDeleted

ListNode\* pNext = pToBeDeleted->m\_pNext;

pToBeDeleted->m\_nKey = pNext->m\_nKey;

pToBeDeleted->m\_pNext = pNext->m\_pNext;

// delete the node next to the pToBeDeleted

delete pNext;

pNext = NULL;

}

// if pToBeDeleted is the last node in the list

else

{

// get the node prior to pToBeDeleted

ListNode\* pNode = pListHead;

while(pNode->m\_pNext != pToBeDeleted)

{

pNode = pNode->m\_pNext;

}

// deleted pToBeDeleted

pNode->m\_pNext = NULL;

delete pToBeDeleted;

pToBeDeleted = NULL;

}

}

值得注意的是，为了让代码看起来简洁一些，上面 的代码基于两个假设：（1）给定的结点的确在链表中；（2）给定的要删除的结点不是链表的头结点。不考虑第一个假设对代码的鲁棒性是有影响的。至于第二个 假设，当整个列表只有一个结点时，代码会有问题。但这个假设不算很过分，因为在有些链表的实现中，会创建一个虚拟的链表头，并不是一个实际的链表结点。这 样要删除的结点就不可能是链表的头结点了。当然，在面试中，我们可以把这些假设和面试官交流。这样，面试官还是会觉得我们考虑问题很周到的。

**程序员面试题精选**（56）：找出两个链表的第一个公共结点收藏

新一篇: 1到n之间1的个数 | 旧一篇: vs下三个比较实用方便的小技巧

题目：两个单向链表，找出它们的第一个公共结点。

链表的结点定义为：

struct ListNode

{

int m\_nKey;

ListNode\* m\_pNext;

};

分析：这是一道微软的面试题。微软非常喜欢与链表相关的题目，因此在微软的面试题中，链表出现的概率相当高。

如果两个单向链表有公共的结点，也就是说两个链 表从某一结点开始，它们的m\_pNext都指向同一个结点。但由于是单向链表的结点，每个结点只有一个m\_pNext，因此从第一个公共结点开始，之后它 们所有结点都是重合的，不可能再出现分叉。所以，两个有公共结点而部分重合的链表，拓扑形状看起来像一个Y，而不可能像X。

看到这个题目，第一反应就是蛮力法：在第一链表 上顺序遍历每个结点。每遍历一个结点的时候，在第二个链表上顺序遍历每个结点。如果此时两个链表上的结点是一样的，说明此时两个链表重合，于是找到了它们 的公共结点。如果第一个链表的长度为m，第二个链表的长度为n，显然，该方法的时间复杂度为O(mn)。

接下来我们试着去寻找一个线性时间复杂度的算 法。我们先把问题简化：如何判断两个单向链表有没有公共结点？前面已经提到，如果两个链表有一个公共结点，那么该公共结点之后的所有结点都是重合的。那 么，它们的最后一个结点必然是重合的。因此，我们判断两个链表是不是有重合的部分，只要分别遍历两个链表到最后一个结点。如果两个尾结点是一样的，说明它 们用重合；否则两个链表没有公共的结点。

在上面的思路中，顺序遍历两个链表到尾结点的时 候，我们不能保证在两个链表上同时到达尾结点。这是因为两个链表不一定长度一样。但如果假设一个链表比另一个长l个结点，我们先在长的链表上遍历l个结 点，之后再同步遍历，这个时候我们就能保证同时到达最后一个结点了。由于两个链表从第一个公共结点考试到链表的尾结点，这一部分是重合的。因此，它们肯定 也是同时到达第一公共结点的。于是在遍历中，第一个相同的结点就是第一个公共的结点。

在这个思路中，我们先要分别遍历两个链表得到它们的长度，并求出两个长度之差。在长的链表上先遍历若干次之后，再同步遍历两个链表，知道找到相同的结点，或者一直到链表结束。此时，如果第一个链表的长度为m，第二个链表的长度为n，该方法的时间复杂度为O(m+n)。

基于这个思路，我们不难写出如下的代码：

///////////////////////////////////////////////////////////////////////

// Find the first common node in the list with head pHead1 and

// the list with head pHead2

// Input: pHead1 - the head of the first list

// pHead2 - the head of the second list

// Return: the first common node in two list. If there is no common

// nodes, return NULL

///////////////////////////////////////////////////////////////////////

ListNode\* FindFirstCommonNode( ListNode \*pHead1, ListNode \*pHead2)

{

// Get the length of two lists

unsigned int nLength1 = ListLength(pHead1);

unsigned int nLength2 = ListLength(pHead2);

int nLengthDif = nLength1 - nLength2;

// Get the longer list

ListNode \*pListHeadLong = pHead1;

ListNode \*pListHeadShort = pHead2;

if(nLength2 > nLength1)

{

pListHeadLong = pHead2;

pListHeadShort = pHead1;

nLengthDif = nLength2 - nLength1;

}

// Move on the longer list

for(int i = 0; i < nLengthDif; ++ i)

pListHeadLong = pListHeadLong->m\_pNext;

// Move on both lists

while((pListHeadLong != NULL) &&

(pListHeadShort != NULL) &&

(pListHeadLong != pListHeadShort))

{

pListHeadLong = pListHeadLong->m\_pNext;

pListHeadShort = pListHeadShort->m\_pNext;

}

// Get the first common node in two lists

ListNode \*pFisrtCommonNode = NULL;

if(pListHeadLong == pListHeadShort)

pFisrtCommonNode = pListHeadLong;

return pFisrtCommonNode;

}

///////////////////////////////////////////////////////////////////////

// Get the length of list with head pHead

// Input: pHead - the head of list

// Return: the length of list

///////////////////////////////////////////////////////////////////////

unsigned int ListLength(ListNode\* pHead)

{

unsigned int nLength = 0;

ListNode\* pNode = pHead;

while(pNode != NULL)

{

++ nLength;

pNode = pNode->m\_pNext;

}

return nLength;

}

**程序员面试题精选**100题(31)-从尾到头输出链表

题目：输入一个链表的头结点，从尾到头反过来输出每个结点的值。链表结点定义如下：

struct ListNode

{

int m\_nKey;

ListNode\* m\_pNext;

};

分析：这是一道很有意思的面试题。该题以及它的变体经常出现在各大公司的面试、笔试题中。

看到这道题后，第一反应是从头到尾输出比较简单。于是很自然地想到把链表中链接结点的指针反转过来，改变链表的方向。然后就可以从头到尾输出了。反转链表的算法详见本人面试题精选系列的第19题，在此不再细述。但该方法需要额外的操作，应该还有更好的方法。

接下来的想法是从头到尾遍历链表，每经过一个结点的时候，把该结点放到一个栈中。当遍历完整个链表后，再从栈顶开始输出结点的值，此时输出的结点的顺序已经反转过来了。该方法需要维护一个额外的栈，实现起来比较麻烦。

既然想到了栈来实现这个函数，而递归本质上就是一个栈结构。于是很自然的又想到了用递归来实现。要实现反过来输出链表，我们每访问到一个结点的时候，先递归输出它后面的结点，再输出该结点自身，这样链表的输出结果就反过来了。

基于这样的思路，不难写出如下代码：

///////////////////////////////////////////////////////////////////////

// Print a list from end to beginning

// Input: pListHead - the head of list

///////////////////////////////////////////////////////////////////////

void PrintListReversely(ListNode\* pListHead)

{

if(pListHead != NULL)

{

// Print the next node first

if (pListHead->m\_pNext != NULL)

{

PrintListReversely(pListHead->m\_pNext);

}

// Print this node

printf("%d", pListHead->m\_nKey);

}

}

扩展：该题还有两个常见的变体：

1. 从尾到头输出一个字符串；

2. 定义一个函数求字符串的长度，要求该函数体内不能声明任何变量。

**程序员面试题精选100题(49)-复杂链表的复制[算法]**

题目：有一个复杂链表，其结点除了有一个m\_pNext指针指向下一个结点外，还有一个m\_pSibling指向链表中的任一结点或者NULL。其结点的C++定义如下：

                struct ComplexNode

{

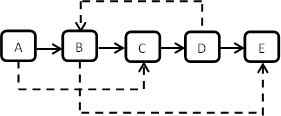
    int m\_nValue;

    ComplexNode\* m\_pNext;

    ComplexNode\* m\_pSibling;

};

                下图是一个含有5个结点的该类型复杂链表。图中实线箭头表示m\_pNext指针，虚线箭头表示m\_pSibling指针。为简单起见，指向NULL的指针没有画出。



请完成函数ComplexNode\* Clone(ComplexNode\* pHead)，以复制一个复杂链表。

                分析：在常见的数据结构上稍加变化，这是一种很新颖的面试题。要在不到一个小时的时间里解决这种类型的题目，我们需要较快的反应能力，对数据结构透彻的理解以及扎实的编程功底。

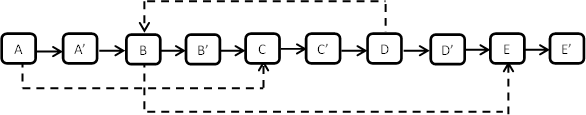
           看到这个问题，我的第一反应是分成两步：第一步是复制原始链表上的每个链表，并用m\_pNext链接起来。第二步，假设原始链表中的某节点N的m\_pSibling指向结点S。由于S的位置在链表上有可能在N的前面也可能在N的后面，所以要定位N的位置我们需要从原始链表的头结点开始找。假设从原始链表的头结点开始经过s步找到结点S。那么在复制链表上结点N的m\_pSibling的S’，离复制链表的头结点的距离也是s。用这种办法我们就能为复制链表上的每个结点设置m\_pSibling了。

                对一个含有n个结点的链表，由于定位每个结点的m\_pSibling，都需要从链表头结点开始经过O(n)步才能找到，因此这种方法的总时间复杂度是O(n2)。

                由于上述方法的时间主要花费在定位结点的m\_pSibling上面，我们试着在这方面去做优化。我们还是分为两步：第一步仍然是复制原始链表上的每个结点N，并创建N’，然后把这些创建出来的结点链接起来。这里我们对<N，N’>的配对信息放到一个哈希表中。第二步还是设置复制链表上每个结点的m\_pSibling。如果在原始链表中结点N的m\_pSibling指向结点S，那么在复制链表中，对应的N’应该指向S’。由于有了哈希表，我们可以用O(1)的时间根据S找到S’。

                第二种方法相当于用空间换时间，以O(n)的空间消耗实现了O(n)的时间效率。

                接着我们来换一种思路，在不用辅助空间的情况下实现O(n)的时间效率。第三种方法的第一步仍然是根据原始链表的每个结点N，创建对应的N’。这一次，我们把N’链接在N的后面。实例中的链表经过这一步之后变成了：



                这一步的代码如下：

                /////////////////////////////////////////////////////////////////////////////////

// Clone all nodes in a complex linked list with head pHead,

// and connect all nodes with m\_pNext link

/////////////////////////////////////////////////////////////////////////////////

void CloneNodes(ComplexNode\* pHead)

{

    ComplexNode\* pNode = pHead;

    while(pNode != NULL)

    {

        ComplexNode\* pCloned = new ComplexNode();

        pCloned->m\_nValue = pNode->m\_nValue;

        pCloned->m\_pNext = pNode->m\_pNext;

        pCloned->m\_pSibling = NULL;

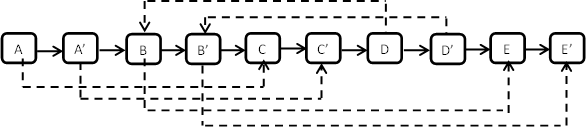
        pNode->m\_pNext = pCloned;

        pNode = pCloned->m\_pNext;

    }

}

                第二步是设置我们复制出来的链表上的结点的m\_pSibling。假设原始链表上的N的m\_pSibling指向结点S，那么其对应复制出来的N’是N->m\_pNext，同样S’也是S->m\_pNext。这就是我们在上一步中把每个结点复制出来的结点链接在原始结点后面的原因。有了这样的链接方式，我们就能在O(1)中就能找到每个结点的m\_pSibling了。例子中的链表经过这一步，就变成如下结构了：



 这一步的代码如下：

                /////////////////////////////////////////////////////////////////////////////////

// Connect sibling nodes in a complex link list

/////////////////////////////////////////////////////////////////////////////////

void ConnectSiblingNodes(ComplexNode\* pHead)

{

    ComplexNode\* pNode = pHead;

    while(pNode != NULL)

    {

        ComplexNode\* pCloned = pNode->m\_pNext;

        if(pNode->m\_pSibling != NULL)

        {

            pCloned->m\_pSibling = pNode->m\_pSibling->m\_pNext;

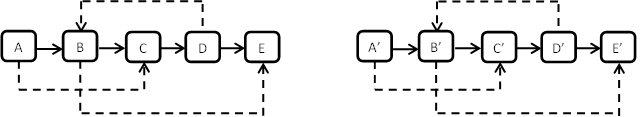
        }

        pNode = pCloned->m\_pNext;

    }

}

                第三步是把这个长链表拆分成两个：把奇数位置的结点链接起来就是原始链表，把偶数位置的结点链接出来就是复制出来的链表。上述例子中的链表拆分之后的两个链表如下：



                要实现这一步，也不是很难的事情。其对应的代码如下：

                /////////////////////////////////////////////////////////////////////////////////

// Split a complex list into two:

// Reconnect nodes to get the original list, and its cloned list

/////////////////////////////////////////////////////////////////////////////////

ComplexNode\* ReconnectNodes(ComplexNode\* pHead)

{

    ComplexNode\* pNode = pHead;

    ComplexNode\* pClonedHead = NULL;

    ComplexNode\* pClonedNode = NULL;

    if(pNode != NULL)

    {

        pClonedHead = pClonedNode = pNode->m\_pNext;

        pNode->m\_pNext = pClonedNode->m\_pNext;

        pNode = pNode->m\_pNext;

    }

    while(pNode != NULL)

    {

        pClonedNode->m\_pNext = pNode->m\_pNext;

        pClonedNode = pClonedNode->m\_pNext;

        pNode->m\_pNext = pClonedNode->m\_pNext;

        pNode = pNode->m\_pNext;

    }

    return pClonedHead;

}

                我们把上面三步合起来，就是复制链表的完整过程：

/////////////////////////////////////////////////////////////////////////////////

// Clone a complex linked list with head pHead

/////////////////////////////////////////////////////////////////////////////////

ComplexNode\* Clone(ComplexNode\* pHead)

{

    CloneNodes(pHead);

    ConnectSiblingNodes(pHead);

    return ReconnectNodes(pHead);

}