轻松搞定面试中的链表题目 - 计算机的艺术 - CSDN博客

星期六, 三月 23, 2019 9:34 下午

已剪辑自: https://blog.csdn.net/luckyxiaoqiang/article/details/7393134

链表是最基本的数据结构，面试官也常常用链表来考察面试者的基本能力，而且链表相关的操作相对而言比较简单，也适合考察写代码的能力。链表的操作也离不开指针，指针又很容易导致出错。综合多方面的原因，链表题目在面试中占据着很重要的地位。本文对链表相关的面试题做了较为全面的整理，希望能对找工作的同学有所帮助。

链表结点声明如下：

```
struct ListNode
{
  int m_nKey;
  ListNode * m_pNext;

};
```

题目列表：

详细解答

1. 求单链表中结点的个数

这是最最基本的了，应该能够迅速写出正确的代码，注意检查链表是否为空。时间复杂度为O（n）。参考代码如下：

```
1.  // 求单链表中结点的个数
2.  unsigned int GetListLength(ListNode * pHead)
3.  {
4.   if(pHead == NULL)
5.   return 0;
6.
7.   unsigned int nLength = 0;
8.   ListNode * pCurrent = pHead;
9.   while(pCurrent != NULL)
10.  {
11.  nLength++;
12.  pCurrent = pCurrent->m_pNext;
13.  }
14.  return nLength;
15. }
```

2. 将单链表反转

从头到尾遍历原链表，每遍历一个结点，将其摘下放在新链表的最前端。注意链表为空和只有一个结点的情况。时间复杂度为O（n）。参考代码如下：

```
// 反转单链表
ListNode * ReverseList(ListNode * pHead)
{
        // 如果链表为空或只有一个结点，无需反转，直接返回原链表头指针
 if(pHead == NULL || pHead->m_pNext == NULL)
 return pHead;

 ListNode * pReversedHead = NULL; // 反转后的新链表头指针，初始为NULL
 ListNode * pCurrent = pHead;
 while(pCurrent != NULL)
 {
 ListNode * pTemp = pCurrent;
 pCurrent = pCurrent->m_pNext;
 pTemp->m_pNext = pReversedHead; // 将当前结点摘下，插入新链表的最前端
 pReversedHead = pTemp;
 }
 return pReversedHead;
}
```

3. 查找单链表中的倒数第K个结点（k > 0）

最普遍的方法是，先统计单链表中结点的个数，然后再找到第（n-k）个结点。注意链表为空，k为0，k为1，k大于链表中节点个数时的情况。时间复杂度为O（n）。代码略。

这里主要讲一下另一个思路，这种思路在其他题目中也会有应用。

主要思路就是使用两个指针，先让前面的指针走到正向第k个结点，这样前后两个指针的距离差是k-1，之后前后两个指针一起向前走，前面的指针走到最后一个结点时，后面指针所指结点就是倒数第k个结点。

参考代码如下：

```
// 查找单链表中倒数第K个结点
ListNode * RGetKthNode(ListNode * pHead, unsigned int k) // 函数名前面的R代表反向
{
 if(k == 0 || pHead == NULL) // 这里k的计数是从1开始的，若k为0或链表为空返回NULL
 return NULL;

 ListNode * pAhead = pHead;
 ListNode * pBehind = pHead;
 while(k > 1 && pAhead != NULL) // 前面的指针先走到正向第k个结点
 {
 pAhead = pAhead->m_pNext;
 k--;
 }
 if(k > 1 || pAhead == NULL)     // 结点个数小于k，返回NULL
 return NULL;
 while(pAhead->m_pNext != NULL)  // 前后两个指针一起向前走，直到前面的指针指向最后一个结点
 {
 pBehind = pBehind->m_pNext;
 pAhead = pAhead->m_pNext;
 }
 return pBehind;  // 后面的指针所指结点就是倒数第k个结点
}
```

4. 查找单链表的中间结点

此题可应用于上一题类似的思想。也是设置两个指针，只不过这里是，两个指针同时向前走，前面的指针每次走两步，后面的指针每次走一步，前面的指针走到最后一个结点时，后面的指针所指结点就是中间结点，即第（n/2+1）个结点。注意链表为空，链表结点个数为1和2的情况。时间复杂度O（n）。参考代码如下：

```
// 获取单链表中间结点，若链表长度为n(n>0)，则返回第n/2+1个结点
ListNode * GetMiddleNode(ListNode * pHead)
{
 if(pHead == NULL || pHead->m_pNext == NULL) // 链表为空或只有一个结点，返回头指针
 return pHead;

 ListNode * pAhead = pHead;
 ListNode * pBehind = pHead;
 while(pAhead->m_pNext != NULL) // 前面指针每次走两步，直到指向最后一个结点，后面指针每次走一步
 {
 pAhead = pAhead->m_pNext;
 pBehind = pBehind->m_pNext;
 if(pAhead->m_pNext != NULL)
 pAhead = pAhead->m_pNext;
 }
 return pBehind; // 后面的指针所指结点即为中间结点
}
```

5. 从尾到头打印单链表

对于这种颠倒顺序的问题，我们应该就会想到栈，后进先出。所以，这一题要么自己使用栈，要么让系统使用栈，也就是递归。注意链表为空的情况。时间复杂度为O（n）。参考代码如下：

自己使用栈：

```
// 从尾到头打印链表，使用栈
void RPrintList(ListNode * pHead)
{
 std::stack<ListNode *> s;
 ListNode * pNode = pHead;
 while(pNode != NULL)
 {
 s.push(pNode);
 pNode = pNode->m_pNext;
 }
 while(!s.empty())
 {
 pNode = s.top();
 printf("%d\t", pNode->m_nKey);
 s.pop();
 }
}
```

使用递归函数：

```
// 从尾到头打印链表，使用递归
void RPrintList(ListNode * pHead)
{
 if(pHead == NULL)
 {
 return;
 }
 else
 {
 RPrintList(pHead->m_pNext);
 printf("%d\t", pHead->m_nKey);
 }
```

```
}
```

6. 已知两个单链表pHead1 和pHead2 各自有序，把它们合并成一个链表依然有序

这个类似归并排序。尤其注意两个链表都为空，和其中一个为空时的情况。只需要O（1）的空间。时间复杂度为O（max(len1, len2)）。参考代码如下：

```
// 合并两个有序链表
ListNode * MergeSortedList(ListNode * pHead1, ListNode * pHead2)
{
 if(pHead1 == NULL)
 return pHead2;
 if(pHead2 == NULL)
 return pHead1;
 ListNode * pHeadMerged = NULL;
 if(pHead1->m_nKey < pHead2->m_nKey)
 {
 pHeadMerged = pHead1;
 pHeadMerged->m_pNext = NULL;
 pHead1 = pHead1->m_pNext;
 }
 else
 {
 pHeadMerged = pHead2;
 pHeadMerged->m_pNext = NULL;
 pHead2 = pHead2->m_pNext;
 }
 ListNode * pTemp = pHeadMerged;
 while(pHead1 != NULL && pHead2 != NULL)
 {
 if(pHead1->m_nKey < pHead2->m_nKey)
 {
 pTemp->m_pNext = pHead1;
 pHead1 = pHead1->m_pNext;
 pTemp = pTemp->m_pNext;
 pTemp->m_pNext = NULL;
 }
 else
 {
 pTemp->m_pNext = pHead2;
 pHead2 = pHead2->m_pNext;
 pTemp = pTemp->m_pNext;
 pTemp->m_pNext = NULL;
 }
 }
 if(pHead1 != NULL)
 pTemp->m_pNext = pHead1;
 else if(pHead2 != NULL)
 pTemp->m_pNext = pHead2;
 return pHeadMerged;
}
```

也有如下递归解法：

```
ListNode * MergeSortedList(ListNode * pHead1, ListNode * pHead2)
{
 if(pHead1 == NULL)
 return pHead2;
 if(pHead2 == NULL)
 return pHead1;
 ListNode * pHeadMerged = NULL;
 if(pHead1->m_nKey < pHead2->m_nKey)
 {
 pHeadMerged = pHead1;
 pHeadMerged->m_pNext = MergeSortedList(pHead1->m_pNext, pHead2);
 }
 else
 {
 pHeadMerged = pHead2;
 pHeadMerged->m_pNext = MergeSortedList(pHead1, pHead2->m_pNext);
 }
 return pHeadMerged;
}
```

7. 判断一个单链表中是否有环

这里也是用到两个指针。如果一个链表中有环，也就是说用一个指针去遍历，是永远走不到头的。因此，我们可以用两个指针去遍历，一个指针一次走两步，一个指针一次走一步，如果有环，两个指针肯定会在环中相遇。时间复杂度为O（n）。参考代码如下：

```
bool HasCircle(ListNode * pHead)
{
 ListNode * pFast = pHead; // 快指针每次前进两步
 ListNode * pSlow = pHead; // 慢指针每次前进一步
 while(pFast != NULL && pFast->m_pNext != NULL)
 {
 pFast = pFast->m_pNext->m_pNext;
 pSlow = pSlow->m_pNext;
 if(pSlow == pFast) // 相遇，存在环
 return true;
 }
 return false;
}
```

8. 判断两个单链表是否相交

如果两个链表相交于某一节点，那么在这个相交节点之后的所有节点都是两个链表所共有的。也就是说，如果两个链表相交，那么最后一个节点肯定是共有的。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。时间复杂度为O(len1+len2)，因为只需要一个额外指针保存最后一个节点地址，空间复杂度为O(1)。参考代码如下：

```
bool IsIntersected(ListNode * pHead1, ListNode * pHead2)
{
        if(pHead1 == NULL || pHead2 == NULL)
                return false;

 ListNode * pTail1 = pHead1;
 while(pTail1->m_pNext != NULL)
 pTail1 = pTail1->m_pNext;

 ListNode * pTail2 = pHead2;
 while(pTail2->m_pNext != NULL)
 pTail2 = pTail2->m_pNext;
 return pTail1 == pTail2;
}
```

9. 求两个单链表相交的第一个节点

对第一个链表遍历，计算长度len1，同时保存最后一个节点的地址。
对第二个链表遍历，计算长度len2，同时检查最后一个节点是否和第一个链表的最后一个节点相同，若不相同，不相交，结束。
两个链表均从头节点开始，假设len1大于len2，那么将第一个链表先遍历len1-len2个节点，此时两个链表当前节点到第一个相交节点的距离就相等了，然后一起向后遍历，知道两个节点的地址相同。

时间复杂度，O(len1+len2)。参考代码如下：

```
ListNode* GetFirstCommonNode(ListNode * pHead1, ListNode *
pHead2)
{
 if(pHead1 == NULL || pHead2 == NULL)
 return NULL;

 int len1 = 1;
 ListNode * pTail1 = pHead1;
 while(pTail1->m_pNext != NULL)
 {
 pTail1 = pTail1->m_pNext;
 len1++;
 }

 int len2 = 1;
 ListNode * pTail2 = pHead2;
 while(pTail2->m_pNext != NULL)
 {
 pTail2 = pTail2->m_pNext;
 len2++;
 }

 if(pTail1 != pTail2) // 不相交直接返回NULL
 return NULL;

 ListNode * pNode1 = pHead1;
 ListNode * pNode2 = pHead2;
       // 先对齐两个链表的当前结点，使之到尾节点的距离相等
 if(len1 > len2)
 {
 int k = len1 - len2;
 while(k--)
 pNode1 = pNode1->m_pNext;
 }
 else
 {
 int k = len2 - len1;
 while(k--)
 pNode2 = pNode2->m_pNext;
 }
 while(pNode1 != pNode2)
 {
 pNode1 = pNode1->m_pNext;
 pNode2 = pNode2->m_pNext;
 }
       return pNode1;
}
```

10. 已知一个单链表中存在环，求进入环中的第一个节点

首先判断是否存在环，若不存在结束。在环中的一个节点处断开（当然函数结束时不能破坏原链表），这样就形成了两个相交的单链表，求进入环中的第一个节点也就转换成了求两个单链表相交的第一个节点。参考代码如下：

```
ListNode* GetFirstNodeInCircle(ListNode * pHead)
{
 if(pHead == NULL || pHead->m_pNext == NULL)
 return NULL;

 ListNode * pFast = pHead;
 ListNode * pSlow = pHead;
 while(pFast != NULL && pFast->m_pNext != NULL)
 {
 pSlow = pSlow->m_pNext;
 pFast = pFast->m_pNext->m_pNext;
 if(pSlow == pFast)
 break;
 }
 if(pFast == NULL || pFast->m_pNext == NULL)
 return NULL;

 // 将环中的此节点作为假设的尾节点，将它变成两个单链表相交问题
 ListNode * pAssumedTail = pSlow;
 ListNode * pHead1 = pHead;
 ListNode * pHead2 = pAssumedTail->m_pNext;

 ListNode * pNode1, * pNode2;
 int len1 = 1;
 ListNode * pNode1 = pHead1;
 while(pNode1 != pAssumedTail)
 {
 pNode1 = pNode1->m_pNext;
 len1++;
 }

 int len2 = 1;
 ListNode * pNode2 = pHead2;
 while(pNode2 != pAssumedTail)
 {
 pNode2 = pNode2->m_pNext;
 len2++;
 }

 pNode1 = pHead1;
 pNode2 = pHead2;
 // 先对齐两个链表的当前结点，使之到尾节点的距离相等
 if(len1 > len2)
 {
 int k = len1 - len2;
 while(k--)
 pNode1 = pNode1->m_pNext;
 }
 else
 {
 int k = len2 - len1;
 while(k--)
 pNode2 = pNode2->m_pNext;
 }
 while(pNode1 != pNode2)
 {
 pNode1 = pNode1->m_pNext;
 pNode2 = pNode2->m_pNext;
 }
   return pNode1;
}
```

11. 给出一单链表头指针pHead和一节点指针pToBeDeleted，O(1)时间复杂度删除节点pToBeDeleted

对于删除节点，我们普通的思路就是让该节点的前一个节点指向该节点的下一个节点，这种情况需要遍历找到该节点的前一个节点，时间复杂度为O(n)。对于链表，链表中的每个节点结构都是一样的，所以我们可以把该节点的下一个节点的数据复制到该节点，然后删除下一个节点即可。要注意最后一个节点的情况，这个时候只能用常见的方法来操作，先找到前一个节点，但总体的平均时间复杂度还是O(1)。参考代码如下：

```
1.  void Delete(ListNode * pHead, ListNode * pToBeDeleted)
2.  {
3.   if(pToBeDeleted == NULL)
4.   return;
5.   if(pToBeDeleted->m_pNext != NULL)
6.   {
7.   pToBeDeleted->m_nKey = pToBeDeleted->m_pNext->m_nKey; // 将下一个
    节点的数据复制到本节点，然后删除下一个节点
8.   ListNode * temp = pToBeDeleted->m_pNext;
9.   pToBeDeleted->m_pNext = pToBeDeleted->m_pNext->m_pNext;
10.  delete temp;
11.  }
12.  else // 要删除的是最后一个节点
13.  {
14.  if(pHead == pToBeDeleted) // 链表中只有一个节点的情况
15.  {
16.  pHead = NULL;
17.  delete pToBeDeleted;
18.  }
19.  else
20.  {
21.  ListNode * pNode = pHead;
22.  while(pNode->m_pNext != pToBeDeleted) // 找到倒数第二个节点
23.  pNode = pNode->m_pNext;
24.  pNode->m_pNext = NULL;
25.  delete pToBeDeleted;
26.  }
27.  }
28. }
```