**Linked List question and answer for GS**

# Problem 1: Remove Duplicates from a Linked List

Given an unsorted linked list, and without using a temporary buffer, write a method that will delete any duplicates from the linked list.

|  |  |
| --- | --- |
| **Analysis:**  typedef struct ListNode{ | |
| int data; |

|  |  |
| --- | --- |
| struct ListNode \*next; | |
| }; |

|  |
| --- |
|  |
| int AddElement(struct ListNode \*\*head, int d){ | |

|  |  |
| --- | --- |
| struct ListNode \*newNode = new struct ListNode; | |
| if(newNode == NULL) |

|  |
| --- |
| return 0; |
| struct ListNode \*t = \*head; | |

|  |
| --- |
| newNode->data = d; |
| newNode->next = NULL; | |

|  |
| --- |
| if(\*head == NULL){ |
| \*head = newNode; | |

|  |  |
| --- | --- |
| return 1; | |
| } |

|  |  |
| --- | --- |
| while(t->next != NULL){ | |
| t = t->next; |

|  |
| --- |
| } |
| t->next = newNode; | |

|  |  |
| --- | --- |
| return 1; | |
| } |

|  |
| --- |
|  |
| int RemoveDuplicates(struct ListNode \*head){ | |

|  |
| --- |
| struct ListNode \*current; |
| struct ListNode \*previous; | |

|  |
| --- |
| struct ListNode \*itr; |
| struct ListNode \*tmp; |

|  |  |
| --- | --- |
| if(head == NULL) | |
| return 0; |

|  |  |
| --- | --- |
| if(head->next == NULL) | |
| return 1; |

|  |  |
| --- | --- |
| current = head->next; | |
| previous = head; |

|  |  |
| --- | --- |
| while(current != NULL){ | |
| itr = head; |

|  |
| --- |
| while(itr != current){ |
| // remove node if equal | |

|  |  |
| --- | --- |
| if(itr->data == current->data){ | |
| tmp = current; |

|  |
| --- |
| current = current->next; |
| previous->next = current; | |

|  |  |
| --- | --- |
| delete tmp; | |
| break; |

|  |
| --- |
| } |
| itr = itr->next; | |

|  |
| --- |
| } |
| if(itr == current){ | |

|  |
| --- |
| current = current->next; |
| previous = previous->next; | |

|  |  |
| --- | --- |
| } | |
| } |

|  |
| --- |
| } |
| int main(int argc, char\* argv[]){ | |

|  |  |
| --- | --- |
| struct ListNode \*head = NULL; | |
| struct ListNode \*ptr; |

|  |
| --- |
| AddElement(&head, 3); |
| AddElement(&head, 2); |

|  |
| --- |
| AddElement(&head, 1); |
| AddElement(&head, 2); |

|  |
| --- |
| AddElement(&head, 2); |
| AddElement(&head, 3); |

|  |
| --- |
| AddElement(&head, 1); |
| AddElement(&head, 5); |

|  |  |
| --- | --- |
| cout << "Original List" << endl; | |
| ptr = head; |

|  |
| --- |
| while(ptr!=NULL){ |
| cout << ptr->data << endl; | |

|  |  |
| --- | --- |
| ptr = ptr->next; | |
| } |

|  |
| --- |
| RemoveDuplicates(head); |
| cout << "After removing duplicates " << endl; | |

|  |
| --- |
| ptr = head; |
| while(ptr!=NULL){ | |

|  |  |
| --- | --- |
| cout << ptr->data << endl; | |
| ptr = ptr->next; |

|  |
| --- |
| } |
| return 0; | |

|  |
| --- |
| } |

Complexity: O(n^2)

If you can sort the list in O(nlogn) then it will take O(nlogn)

**Problem 2: Get the Kthnode from end of a linked list. It counts from 1 here, so the 1st node from end is the tail of list.**

For instance, given a linked list with 6 nodes, whose value are 1, 2, 3, 4, 5, 6, its 3rd node from end is the node with value 4.

A node in the list is defined as:

struct ListNode

{

int       m\_nValue;

ListNode\* m\_pNext;

};

**Analysis:**In a list with n nodes, its kthnode from end should be the (n-k+1)thnode from its head. Therefore, if we know the number of nodes n in a list, we can get the required node with n-k+1 steps from its head. How to get the number n? It is easy if we scan the whole list from beginning to end.

The solution above needs to scan a list twice: We get the total number of nodes with the first scan, and reach the kth node from end with the second scan. Unfortunately, interviewers usually expect a solution which only scans a list once.

We have a better solution to get the kth node from end with two pointers. Firstly we move a pointer (denoted as P1) k-1 steps beginning from the head of a list. And then we move another pointer (denoted as P2) beginning from the head, and continue moving the P1 forward at same speed. Since the distance of these two pointers is always k-1, P2 reaches the kth node from end when P1 reaches the tail of a list. It scans a list only once, and it is more efficient than the previous solution.

**The sample code of the solutions with two pointers is shown below:**

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

{

if(pListHead == NULL || k == 0)

return NULL;

ListNode \*pAhead = pListHead;

ListNode \*pBehind = NULL;

for(unsigned int i = 0; i < k - 1; ++ i)

{

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

pAhead = pAhead->m\_pNext;

else

{

return NULL;

}

}

pBehind = pListHead;

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

{

pAhead = pAhead->m\_pNext;

pBehind = pBehind->m\_pNext;

}

return pBehind;

}

**Problem 3: - Reverse a Linked List…??**

**Problem:** Implement a function to reverse a linked list, and return the head of the reversed list. A list node is defined as below:

struct ListNode

{

int       m\_nKey;

ListNode\* m\_pNext;

};

**Analysis:**Lots of pointer operations are necessary to solve problems related to linked lists. Interviewers know that many candidates are prone to make mistakes on pointer operations, so they like problems of linked list to qualify candidates’ programming abilities. During interviews, we had better analyze and design carefully rather than begin to code hastily. It is much better to write robust code with comprehensive analysis than write code quickly with many errors.

Direction of pointers should be adjusted in order to reverse a linked list. We may utilize figures to analyze visually the complex steps to adjust pointers. As shown in the list in Figure 1-a, node h, i and j are three adjacent nodes. Let us assume pointers of all nodes prior to h have been reversed after some operations and all m\_pNext point to their previous nodes. We are going to reverse them\_pNext pointer in node i.

It is noticeable that m\_pNext in node i points to its previous node h, the list is broken and we cannot visit the node j anymore. We should save the node j before the m\_pNext pointer of node i is adjusted to prevent the list becoming broken.

When we adjust the pointer in node i, we need to access to node h since m\_pNext of node i is adjusted to point to node h. Meanwhile, we also need to access to node j because it is necessary to save it otherwise the list will be broken. Therefore, three pointers should be declared in our code, which point to the current visited node, its previous node and its next node.

Lastly we should get the head node of the reversed list. Obviously head in the reversed list should be tail of the original list. Which pointer is tail? It should be a node whose m\_pNext is NULL.

With comprehensive analysis above, we are ready to write code, which is shown below:

ListNode\* ReverseList(ListNode\* pHead)

{

ListNode\* pReversedHead = NULL;

ListNode\* pNode = pHead;

ListNode\* pPrev = NULL;

while(pNode != NULL)

{

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

if(pNext == NULL)

pReversedHead = pNode;

pNode->m\_pNext = pPrev;

pPrev = pNode;

pNode = pNext;

}

return pReversedHead;

}

**Problem 4: - Add on Lists …???**

**Problem:** Nodes in a list represent a number.

**Analysis:** Usually numbers are added beginning from the least significant digits (The digit 3 in the number 123, and the digit 7 in the number 4567). As shown in Figure 1, the least significant digits are at the tail of lists, and they can be accessed after the whole lists are scanned. Therefore, lists should be reversed at first, in order get the least significant digits before other digits. The two reversed lists of lists in Figure 1 are shown in Figure 2.

After two lists are reversed, we can add nodes along the links between nodes, and then reversed the result list after all nodes are added. Therefore, the overall structure to add numbers in two lists can be implemented with the following code in C/C++:

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

{

if(pHead1 == NULL || pHead2 == NULL)

return NULL;

pHead1 = Reverse(pHead1);

pHead2 = Reverse(pHead2);

ListNode\* pResult = AddReversed(pHead1, pHead2);

return Reverse(pResult);

}

Now let’s implement the function AddReversed, to add nodes in two reversed lists. Digits are gotten in nodes along links between nodes. When we get two digits in two lists, we add them and create a new node to store the sum, and append the new node into the list for result. There are two issues worthy of attention: (1) The length of two lists might be different; (2) The sum of two digits may be greater than 10, so we have to take care of the carry when adding two digits. The function AddReversed can be implemented with the following C/C++ code:

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

{

int carry = 0;

ListNode\* pPrev = NULL;

ListNode\* pHead = NULL;

while(pHead1 != NULL || pHead2 != NULL)

{

ListNode\* pNode = AddNode(pHead1, pHead2, &carry);

AppendNode(&pHead, &pPrev, pNode);

if(pHead1 != NULL)

pHead1 = pHead1->m\_pNext;

if(pHead2 != NULL)

pHead2 = pHead2->m\_pNext;

}

if(carry > 0)

{

ListNode\* pNode = CreateListNode(carry);

AppendNode(&pHead, &pPrev, pNode);

}

return pHead;

}

The function AddNode adds digits in two nodes. The third parameter of this function takes the carry for addition calculation, as listed below:

ListNode\* AddNode(ListNode\* pNode1, ListNode\* pNode2, int\* carry)

{

int num1 = 0;

if(pNode1 != NULL)

num1 = pNode1->m\_nValue;

int num2 = 0;

if(pNode2 != NULL)

num2 = pNode2->m\_nValue;

int sum = num1 + num2 + \*carry;

\*carry = (sum >= 10) ? 1 : 0;

int value = (sum >= 10) ? (sum - 10) : sum;

return CreateListNode(value);

}

The function AppendNode is used append a node into the tail of lists. In order to avoid scanning the whole list to get the previous tail every time, the previous tails is stored in the parameter/variable pPrev, as listed in the following code:

void AppendNode(ListNode\*\* pHead, ListNode\*\* pPrev, ListNode\* pNode)

{

if(\*pHead == NULL)

\*pHead = pNode;

if(\*pPrev == NULL)

\*pPrev = pNode;

else

{

(\*pPrev)->m\_pNext = pNode;

\*pPrev = pNode;

}

}

The function CreateListNode is to create a list node according to a value, which is omitted here because it’s quite straightforward.

**Problem 5:** - **Loop in List??**

**Question 1**: How to check whether there is a loop in a linked list?

A node in list is defined as the following structure:

struct ListNode

{

int       m\_nValue;

ListNode\* m\_pNext;

};

**Analysis**: It is a popular interview question. Similar to the problem to get the [Kth node from end](http://codercareer.blogspot.com/2011/10/no-10-k-th-node-from-end.html) is a list, it has a solution with two pointers.

Two pointers are initialized at the head of list. One pointer forwards once at each step, and the other forwards twice at each step. If the faster pointer meets the slower one again, there is a loop in the list. Otherwise there is no loop if the faster one reaches the end of list.

The sample code below is implemented according to this solution. The faster pointer is pFast, and the slower one is pSlow.

bool HasLoop(ListNode\* pHead)

{

if(pHead == NULL)

return false;

ListNode\* pSlow = pHead->m\_pNext;

if(pSlow == NULL)

return false;

ListNode\* pFast = pSlow->m\_pNext;

while(pFast != NULL && pSlow != NULL)

{

if(pFast == pSlow)

return true;

pSlow = pSlow->m\_pNext;

pFast = pFast->m\_pNext;

if(pFast != NULL)

pFast = pFast->m\_pNext;

}

return false;

}

**Question 2**: If there is a loop in a linked list, how to get the entry node of the loop? The entry node is the first node in the loop from head of list. For instance, the entry node of loop in the list of Figure 1 is the node with value 3.

**Analysis**: Inspired by the solution of the first problem, we can also solve this problem with two pointers.

Two pointers are initialized at the head of a list. If there are *n* nodes in the loop, the first pointer forwards *n* steps firstly. And then they forward together, at same speed. When the second pointer reaches the entry node of loop, the first one travels around the loop and returns back to entry node.

The only problem is how to get the numbers in a loop. Let go back to the solution of the first question. We define two pointers, and the faster one meets the slower one if there is a loop. Actually, the meeting node should be inside the loop. Therefore, we can move forward from the meeting node and get the number of nodes in the loop when we arrive at the meeting node again.

The following function MeetingNode gets the meeting node of two pointers if there is a loop in a list, which is a minor modification of the previous HasLoop:

ListNode\* MeetingNode(ListNode\* pHead)

{

if(pHead == NULL)

return NULL;

ListNode\* pSlow = pHead->m\_pNext;

if(pSlow == NULL)

return NULL;

ListNode\* pFast = pSlow->m\_pNext;

while(pFast != NULL && pSlow != NULL)

{

if(pFast == pSlow)

return pFast;

pSlow = pSlow->m\_pNext;

pFast = pFast->m\_pNext;

if(pFast != NULL)

pFast = pFast->m\_pNext;

}

return NULL;

}

We can get the number of nodes in a loop of a list, and the entry node of loop after we know the meeting node, as shown below:

ListNode\* EntryNodeOfLoop(ListNode\* pHead)

{

ListNode\* meetingNode = MeetingNode(pHead);

if(meetingNode == NULL)

return NULL;

// get the number of nodes in loop

int nodesInLoop = 1;

ListNode\* pNode1 = meetingNode;

while(pNode1->m\_pNext != meetingNode)

{

pNode1 = pNode1->m\_pNext;

++nodesInLoop;

}

// move pNode1

pNode1 = pHead;

for(int i = 0; i < nodesInLoop; ++i)

pNode1 = pNode1->m\_pNext;

// move pNode1 and pNode2

ListNode\* pNode2 = pHead;

while(pNode1 != pNode2)

{

pNode1 = pNode1->m\_pNext;

pNode2 = pNode2->m\_pNext;

}

return pNode1;

}

**Problem 6: - Add on Lists??**

**Problem:** Nodes in a list represent a number.

**Analysis:** Usually numbers are added beginning from the least significant digits.

After two lists are reversed, we can add nodes along the links between nodes, and then reversed the result list after all nodes are added. Therefore, the overall structure to add numbers in two lists can be implemented with the following code in C/C++:

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

{

if(pHead1 == NULL || pHead2 == NULL)

return NULL;

pHead1 = Reverse(pHead1);

pHead2 = Reverse(pHead2);

ListNode\* pResult = AddReversed(pHead1, pHead2);

return Reverse(pResult);

}

Now let’s implement the function AddReversed, to add nodes in two reversed lists. Digits are gotten in nodes along links between nodes. When we get two digits in two lists, we add them and create a new node to store the sum, and append the new node into the list for result.

**There are two issues worthy of attention:**

(1) The length of two lists might be different;

(2) The sum of two digits may be greater than 10, so we have to take care of the carry when adding two digits. The function AddReversed can be implemented with the following C/C++ code:

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

{

int carry = 0;

ListNode\* pPrev = NULL;

ListNode\* pHead = NULL;

while(pHead1 != NULL || pHead2 != NULL)

{

ListNode\* pNode = AddNode(pHead1, pHead2, &carry);

AppendNode(&pHead, &pPrev, pNode);

if(pHead1 != NULL)

pHead1 = pHead1->m\_pNext;

if(pHead2 != NULL)

pHead2 = pHead2->m\_pNext;

}

if(carry > 0)

{

ListNode\* pNode = CreateListNode(carry);

AppendNode(&pHead, &pPrev, pNode);

}

return pHead;

}

The function AddNode adds digits in two nodes. The third parameter of this function takes the carry for addition calculation, as listed below:

ListNode\* AddNode(ListNode\* pNode1, ListNode\* pNode2, int\* carry)

{

int num1 = 0;

if(pNode1 != NULL)

num1 = pNode1->m\_nValue;

int num2 = 0;

if(pNode2 != NULL)

num2 = pNode2->m\_nValue;

int sum = num1 + num2 + \*carry;

\*carry = (sum >= 10) ? 1 : 0;

int value = (sum >= 10) ? (sum - 10) : sum;

return CreateListNode(value);

}

The function AppendNode is used append a node into the tail of lists. In order to avoid scanning the whole list to get the previous tail every time, the previous tails is stored in the parameter/variable pPrev, as listed in the following code:

void AppendNode(ListNode\*\* pHead, ListNode\*\* pPrev, ListNode\* pNode)

{

if(\*pHead == NULL)

\*pHead = pNode;

if(\*pPrev == NULL)

\*pPrev = pNode;

else

{

(\*pPrev)->m\_pNext = pNode;

\*pPrev = pNode;

}

}

The function CreateListNode is to create a list node according to a value, which is omitted here because it’s quite straightforward.