Linked Lists

Arrays are quite useful in all programming languages. However, they also have some limitations. They require creating a new array with bigger size and copying the existing elements from the old array to the new one when the capacity of an array is overrun. Additionally, they have to shift some elements in an array when a new element is inserted because memory of an array is sequentially allocated. Such limitations can be overcome by dynamic structures, such as linked lists.

It is not necessary to know the size of a list when it is created, so it is treated as a dynamic data structure. Rather than allocate memory for all elements when a list is initialized, memory is allocated for each node on demand when it is inserted. The space efficiency of lists is better than arrays because there is no vacant memory in lists.

Memory allocation of a list is not continuous because nodes are inserted dynamically and their memory is not allocated at the same time. It costs O(n) time to get the i^{th} node in a list since it has to traverse nodes one by one starting from the head node. It only takes O(1) time to get the i^{th} element in an array. Therefore, time efficiency to search lists is not as good as for arrays.

Linked lists are the most frequently met data structures during interviews. It only takes about 20 lines of code to create a list, insert a node into a list, or delete a node from a list. Compared to other complex data structures, such as hash tables and graphs, lists are more suitable for interviews due to their moderate code size. Additionally, lots of pointer operations are required to handle a list. Candidates without qualified programming abilities cannot implement complete and robust code related to lists. Moreover, lists are also flexible and challenging interview questions can be constructed with them. Therefore, many interviewers like questions related to lists.

Most lists met during interviews are single-linked lists, where each node has a link to its successor. For example, "Print a List from Tail to Head" (Question 13), "Delete a Node from a List in O(1) Time" (Question 43) , "k^{th} Node from the End" (Question 47), "Reverse Lists" (Question 48), and "First Intersection Node of Two Lists" (Question 82) are all about single-linked lists.

Not only are single-linked lists popular for interviews, but other types of lists are also frequently met:

- Usually the tail node in a single-linked list does not have a successor. If every node in a finite list has a successor, a loop is formed. The section *Loop in List* discusses lists with loops.
- If there is also a link to a predecessor besides a link to a successor in each node of a list, it is a double-linked list. The interview question "Binary Search Trees and Double-Linked Lists" (Question 64) is in this category.
- A complex list is composed if each node has a link to any other node (including the node itself). Please refer to the interview question "Clone Complex Lists" (Question 54) for more details on the complex list.

Print Lists from Tail to Head

Question 13 Please implement a function to print a list from its tail to head.

When meeting this question, many candidates' intuition is to reverse links between all nodes and to reverse the direction of a list. It fulfills the requirement if all nodes in the reversed list are printed from head to tail. However, it has to modify the structure of the input list. Is it OK to do so? It depends on the requirement in an interviewers' minds. You should ask your interviewers to clarify their requirement before you describe your solution.

Tip If you are going to modify the input data, you should ask interviewers whether it is allowed to do so.

Usually, printing is a read-only operation, and it is not allowed to modify the content to be printed. Supposing interviewers disallow to reverse the direction of lists here.

Nodes in a list are linked from the head to the tail, but the printing order is from the tail to the head. That is to say, the first node in the list is the last one to be printed, and the last node is the first one to be printed. It is typical "Last In, First Out," so a stack can help to solve this problem. Every node is pushed into a stack when it is visited. After all nodes are visited and pushed into the stack, they are printed from the top of the stack and popped one by one. The printing order is opposite to the order in the list.

The sample code in Listing 3-17 is based on `stack` in the C++ standard template library.

Listing 3-17. *C++ Code to Print a List Reversely (Iterative Version)*

```
void PrintListReversingly_Iteratively(ListNode* pHead) {
 std::stack<ListNode*> nodes;
 ListNode* pNode = pHead;
 while(pNode != NULL) {
     nodes.push(pNode);
     pNode = pNode->m_pNext;
 }
 while(!nodes.empty()) {
     pNode = nodes.top();
     printf("%d\t", pNode->m_nValue);
     nodes.pop();
 }
}
```

It can be solved with a stack and since recursion is essentially equivalent to stacks, so it can also be solved recursively. To print a list in reverse, the next nodes are printed first when a node is visited, and then the currently visited node is printed. The recursive code is shown in Listing 3-18.

Listing 3-18. *C++ Code to Print a List Reversely (Recursive Version)*

```
   void PrintListReversingly_Recursively(ListNode* pHead) {

    if(pHead != NULL) {

        if (pHead->m_pNext != NULL) {

            PrintListReversingly_Recursively(pHead->m_pNext);

        }

        printf("%d\t", pHead->m_nValue);

    }

}
```

The previous recursive code looks more concise, but it has a limitation: the recursive calls may have too many levels to cause call stack overflow errors when the list is very long. The iterative solution with an explicit stack is more robust. More discussion about recursion and iteration are available in the section *Recursion and Iteration*.
Source Code:

```
013_PrintListsReversely.cpp
```

Test Cases:

- Lists with multiple nodes or only one node
- The head node of a list is `NULL`

Sort Lists

Question 14 Please implement a function to sort a given list.

The most efficient sorting algorithm in general, quicksort, is applicable to arrays because elements in arrays can be accessed in O(1) time based on indexes. It takes more time to locate a node in a list, so we have to utilize other algorithms to sort lists.

Let's take a list with four nodes as an example to analyze the process of the insert sort algorithm (Figure 3-6(a)). A list is split into two parts. The first part contains nodes already sorted, and the second part is not sorted yet. The node pointed by P_1 is the last sorted node, which is initialized to the first node, and the node pointed to by P_2 is the next one to be sorted.

When node 1 is the next node to be sorted, there is only one node (node 2) in the sorted list. Because the value 1 is less than the value in the list head, node 1 becomes the new head of the sorted list, and node 2 is linked to the next node of the previous node 1, which is node 4 (Figure 3-6(b)). Node 2 is still the last node in the sorted list, so it does not move P_1, and only moves P_2 to the next node of node 2.

The next node to be sorted is node 4. It traverses from the head node in order to find an appropriate location in the sorted list. Since the value 4 is greater than the value 2 in the last node of the sorted list, it is not necessary to relink nodes. Node 4 becomes the new last node of the sorted list, so it is pointed to by P_1 (Figure 3-6(c)).

Now the next node to be sorted is node 3. It traverses from the head node again in order to find an appropriate location in the sorted list. Node 3 is linked between node 2 and node 4. Node 4 is still the last node in the sorted list. Since there are no nodes left after node 4, the whole list is sorted (Figure 3-6(d)).

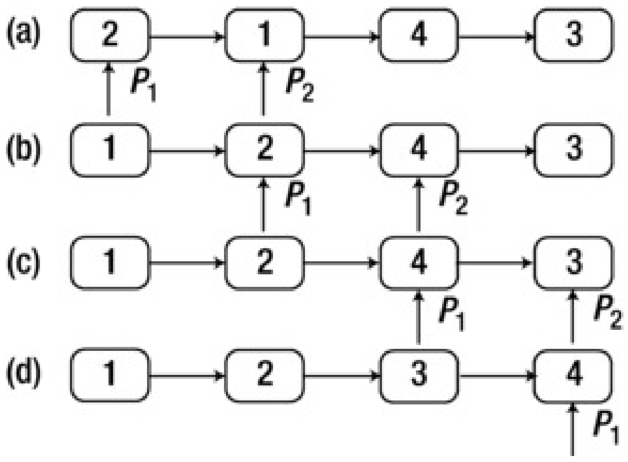

Figure 3-6. *The process to sort a list with four nodes.* P_1 *points to the last node in the sorted list, and* P_2 *points to the next node to be inserted into the sorted list, which always follows* P_1 *. (a)* P_1 *is initialized to the first node, and* P_2 *is initialized to the second one. (b) The node with value 2 is inserted into the sorted list. (c) The node with value 4 is inserted into the sorted list. (d) The node with value 3 is inserted into the sorted list.*

The insert sort algorithm for lists can be implemented in C++, as shown in Listing 3-19.

Listing 3-19. *C++ Code to Sort a List*

```
   void Sort(ListNode** pHead) {
    if(pHead == NULL || *pHead == NULL)

        return;


        ListNode* pLastSorted = *pHead;

    ListNode* pToBeSorted = pLastSorted->m_pNext;

    while(pToBeSorted != NULL) {

        if(pToBeSorted->m_nValue < (*pHead)->m_nValue) {

            pLastSorted->m_pNext = pToBeSorted->m_pNext;

            pToBeSorted->m_pNext = *pHead;

            *pHead = pToBeSorted;

        }

        else {

            ListNode* pNode = *pHead;

            while(pNode != pLastSorted

                && pNode->m_pNext->m_nValue < pToBeSorted->m_nValue) {

                pNode = pNode->m_pNext;

            }


            if(pNode != pLastSorted) {

                pLastSorted->m_pNext = pToBeSorted->m_pNext;

                pToBeSorted->m_pNext = pNode->m_pNext;

                pNode->m_pNext = pToBeSorted;

            }

            else

                pLastSorted = pLastSorted->m_pNext;

        }


        pToBeSorted = pLastSorted->m_pNext;

    }

}
```

Because it has to scan O(n) nodes on the sorted list in order to find an appropriate location to insert a new node, it costs O(n^2) time to sort a list with n nodes. Source Code:

```
014_SortLists.cpp
```

Test Cases:

- Sort a list with multiple nodes, with/without duplicated values
- The input list has only one node
- The input head node of a list is `NULL`

Question 15 Please implement a function to merge two sorted lists into a single sorted list. For example, the merged list of two sorted lists, L_1 and L_2 in Figure 3-7, is L_3.

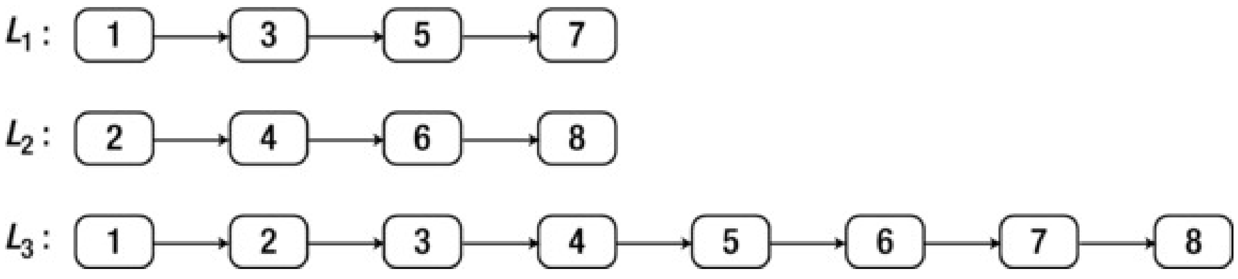

Figure 3-7. *The lists* L$_1$ *and* L$_2$ *are two sorted lists. They compose the list* L$_3$ *when* L$_1$ *and* L$_2$ *are merged.*

The process begins to merge lists from head nodes because traversal on lists begins from head nodes. The head node of L_1 with value 1 is the head node of the merged list because its value is less than the value in the head of node of L_2, as shown in Figure 3-8(a).

It continues to merge the nodes remaining in L_1 and L_2 (the nodes in the rectangles with dashed edges in Figure 3-8). The steps to merge are similar to the ones before because nodes remaining in the two lists are still sorted. It continues to compare values in the head nodes of the two lists. Because the value in the head node of L_2 is less than the value of head node in the remaining L_1, it links the head node of L_2 to the tail of the merged list, as shown in Figure 3-8(b).

When the head node of one list with a lower value than the other is linked to the tail of the merged list, the nodes remaining in two lists are still increasingly sorted, so the steps to merge are the same as before. It is a typical recursive process, so let's implement it based on recursion.

It is important to know when to stop in recursion. It exits recursion when one of these two lists is empty. When L_1 is an empty list, the merged list is L_2. Similarly, the merged list is L_1 if L_2 is empty. If both lists are empty, the merged list is obviously empty.

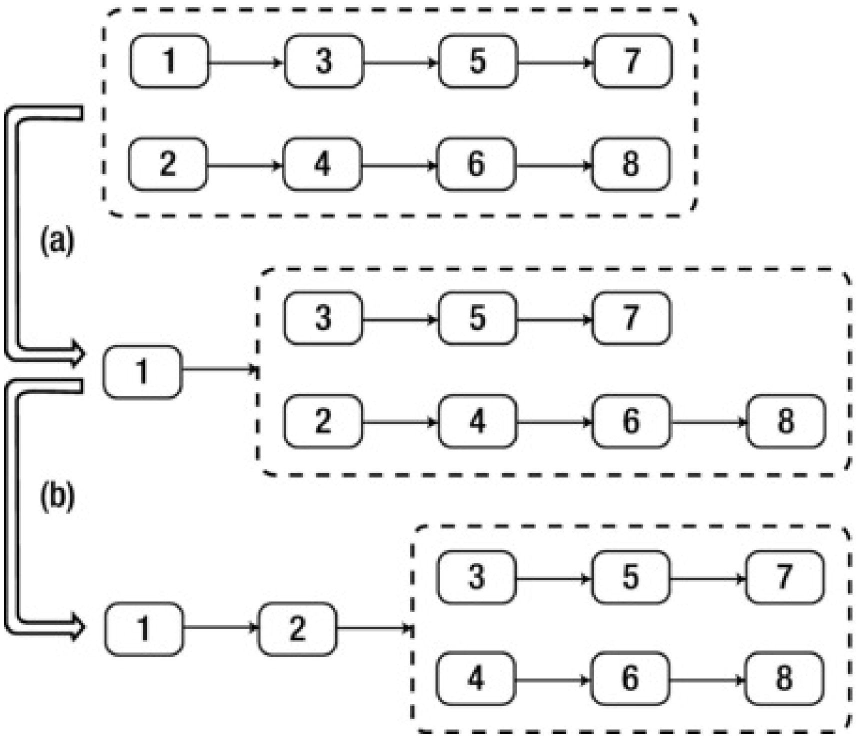

Figure 3-8. *The process to merge two sorted lists. (a) The head of* L_1 *becomes the head of the merged list because the value in the head node of* L_1 *is less than the value in the head node of* L_2 *. (b) The value in the head node of* L_2 *is less than the value in the head node of the remaining* L_1*, so the head node of* L_2 *is connected to the tail of the merged list.*

It is time to write code with careful analysis. A piece of sample code is shown in Listing 3-20.

Listing 3-20. *C++ Code to Merge Sorted List*

```
ListNode* Merge(ListNode* pHead1, ListNode* pHead2) {
    if(pHead1 == NULL)
        return pHead2;
    else if(pHead2 == NULL)
        return pHead1;

    ListNode* pMergedHead = NULL;

    if(pHead1->m_nValue < pHead2->m_nValue) {
        pMergedHead = pHead1;
        pMergedHead->m_pNext = Merge(pHead1->m_pNext, pHead2);
    }
    else {
        pMergedHead = pHead2;
        pMergedHead->m_pNext = Merge(pHead1, pHead2->m_pNext);
    }

    return pMergedHead;
}
```

Sorted lists can also be merged iteratively. Actually, the iterative solution is more robust with long lists. If you feel interested, please try to implement your own iterative version.

Source Code:

```
015_MergeSortedLists.cpp
```

Test Cases:

- There are/are not duplicated values in the two arrays
- One or two input list head nodes are `NULL`

Loop in List

Question 16 How do you check whether there is a loop in a linked list? For example, the list in Figure 3-9 contains a loop.

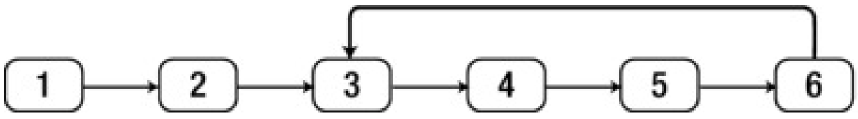

Figure 3-9. *A list with a loop*

This is a popular interview question. It can be solved with two pointers, which are initialized at the head of list. One pointer advances once at each step, and the other advances 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 in Listing 3-21 is implemented based on this solution. The faster pointer is `pFast`, and the slower one is `pSlow`.

Listing 3-21. *C++ Code to Check Whether a List Contains a Loop*

```
  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;
}
```

Source Code:

`016_LoopsInLists.cpp`

Test Cases:

- There is a loop in a list (including cases where there are one/multiple nodes in a loop, or a loop contains all nodes in a list)
- There is not a loop in a list
- The input node of the list head is `NULL`

Question 17 If there is a loop in a linked list, how do you get the entry node of the loop? The entry node is the first node in the loop from the head of a list. For instance, the entry node of the loop in the list of Figure 3-9 is the node with value 3.

Inspired by the solution of the preceding problem, we can also solve this problem with two pointers.

Two pointers P_1 and P_2 are initialized to point to the head of a list. If there are n nodes in the loop, the first pointer move forward n steps, and then two pointers move together with same speed. When the second pointer reaches the entry node of the loop, the first one travels around the loop and returns back to the entry node.

Let's take the list in Figure 3-9 as an example. P_1 and P_2 are first initialized to point to the head node of the list (Figure 3-10(a)). There are four nodes in the loop of the list, so P_1 moves four steps ahead and reaches the node with value 5 (Figure 3-10(b)). Then these two pointers move two steps, and they meet at the node with value 3, which is the entry node of the loop, as shown in Figure 3-10(c).

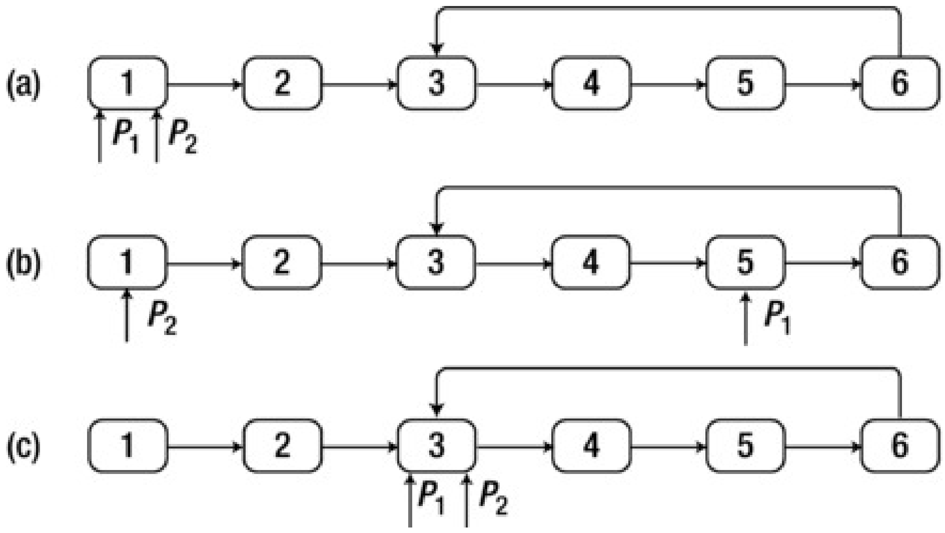

Figure 3-10. *Process to find the entry node of a loop in a list. (a) Pointers* P_1 *and* P_2 *are initialized at the head of list. (b) The pointer* P_1 *moves four steps ahead since there are four nodes in the loop. (c) Both* P_1 *and* P_2 *move ahead till they meet at the entry node of the loop.*

The only problem is how to figure out the number of nodes inside a loop. Let's go back to the solution of the previous question. Two pointers are employed, and the faster one meets the slower one if there is a loop. The meeting node should be inside the loop. Therefore, we can move forward from the meeting node and count nodes in the loop until we arrive at the meeting node again.

The function `MeetingNode` in Listing 3-22 finds the meeting node of two pointers if there is a loop in a list, which is a minor modification of the previous `HasLoop`.

Listing 3-22. *C++ Code to Get a Meeting Node in a Loop*

```
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;
}
```

The function `MeetingNode` returns a node in the loop when there is a loop in the list. Otherwise, it returns `NULL`.

After finding the meeting node, it counts nodes in a loop of a list, as well as finding the entry node of the loop with the sample code, as shown in Listing 3-23.

Listing 3-23. *C++ Code to Get a Meeting Node in a Loop*

```
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;
}
```

Source Code:

`017_EntryNodeInLoopsInLists.cpp`

Test Cases:

- There is a loop in a list (including cases where there are one/multiple nodes in a loop, or a loop contains all nodes in a list)
- There is not a loop in a list
- The input node of the list head is `NULL`