Username: Pralay Patoria **Book:** Coding Interviews: Questions, Analysis & Solutions. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

Robustness

Robust applications are able to check whether inputs are valid and handle invalid inputs appropriately. If applications are not robust, they might behave in unpredictable ways or crash when unexpected things happen, such as cases when users input incorrect user names, files to be opened do not exist, or servers cannot be connected. User experiences with such applications are almost surely disasters.

It is necessary for candidates to pay careful attention to code robustness. Defensive programming is an effective method of improving robustness, which predicts where it is possible to have problems, and handles problems gracefully. For example, it prompts users to check the file name when the file to be opened does not exist, or it tries a backup server when the server is inaccessible. The behavior of a robust application is predictable.

The most simple and effective defensive programming technique during interviews is to check whether inputs are valid at the entry of each function. When a function has a pointer argument, what is the expected behavior when the pointer is `NULL` ? When a function takes a string as an argument, what is the expected behavior when the string is empty? If all problems have been considered in advance, it is a strong demonstration of defensive programming habits and a candidate's ability to develop robust applications.

Not all problems related to robustness are as simple as validating inputs at function entries. It is better to ask questions in the pattern of " how to handle it when … is not …". For instance, it assumes that there are more than *k* nodes in a list for the interview problem " k^{th} Node from End" (Question 47). Candidates should ask themselves what the expected behavior is when the number of nodes is less than *k*. This kind of question helps to find potential problems and handle them in advance.

k^{th} Node from End

Question 47 Please implement a function to get the k^{th} node from tail of a single linked list. For example, if a list has six nodes whose values are 1, 2, 3, 4, 5, 6 in order from head, the third node from tail contains the value 4.

The program can only move forward in a single-linked list from the head node, but it cannot move backward from the tail. Supposing a list has *n* nodes. The k^{th} node from the tail is the $(n\text{-}k\text{+}1)^{th}$ node from the head. It reaches the target node if it moves *n*-*k*+1 steps along links to the next nodes. How do you find the total number of nodes *n* in a list? Just traverse the whole list and count.

As a result, it gets the k^{th} node from the tail after it traverses the list twice: it traverses the whole list to get the number of nodes *n* in the list, and then it moves along the links between nodes for *n*-*k*+1 steps.

This solution works. However, usually interviewers prefer a solution that traverses the list only once.

In order to solve this problem with only one traversal, two pointers are employed. The first pointer (P_1) moves *k*-1 steps from the head of the list. The second pointer (P_2) begins to move from the head and P_1 continues to move. Since the distance between these two pointers keeps at *k*-1, the second pointer reaches the k^{th} node from the tail when the first one reaches the tail of the list.

Figure 5-6 simulates the process of finding the third node from the end of a list with six nodes. The pointer P_1 moves 2 steps (2=3-1) in the list (Figure 5-6(a)). The pointer P_2 is initialized at the head of the list (Figure 5-6(b)). Then these two pointers move at the same speed. When P_1 reaches the tail of the list, P_2 arrives at the third node from the end (Figure 5-6(c)).

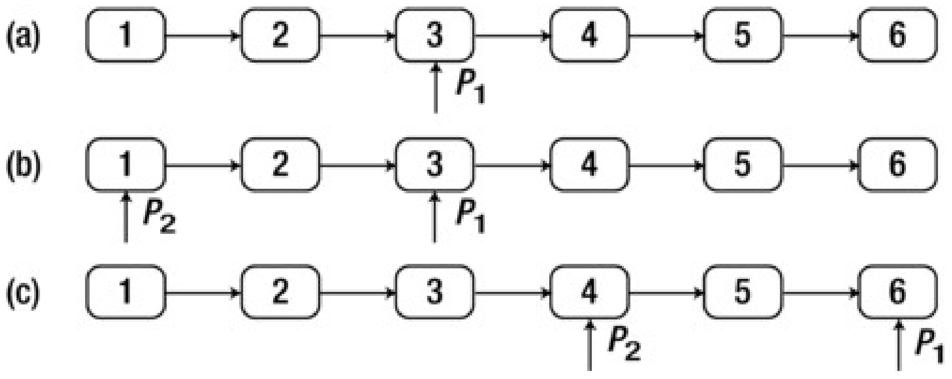

Figure 5-6. *The process to get the third node from the tail of a list with six nodes. (a) Pointer* P_1 *moves two steps on the list. (b) The pointer* P_2 *is initialized at the head of the list. (c) Both* P_1 *and* P_2 *move ahead until* P_1 *reaches the tail of the list. The node pointed to by* P_2 *is the third node from the tail.*

Some candidates implement a solution with two pointers quickly, as shown in Listing 5-19.

Listing 5-19. *C++ Code to Get the* k^{th} *Node from End (Version 1)*

```
ListNode* FindKthToTail(ListNode* pListHead, unsigned int k) {

    ListNode *pAhead = pListHead;

    ListNode *pBehind = NULL;


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

        pAhead = pAhead->m_pNext;

    }
    pBehind = pListHead;

    while(pAhead->m_pNext != NULL){

        pAhead = pAhead->m_pNext;

        pBehind = pBehind->m_pNext;

    }


    return pBehind;

}
```

Many candidates read web pages about the solution with two pointers during interview preparation. When they meet such a problem in interviews, they feel lucky and write code similar to the code in Listing 5-19 quickly. Unfortunately, they may receive a rejection letter rather than an offer. Is it unfair? It is a fair result because the code has three potential risks for crash:

- The head of the list `pListHead` is a `NULL` pointer. An application crashes when it tries to access the `NULL` address in memory.
- The number of nodes in the list with head `pListHead` is less than *k*. It crashes because it tries to move *k*-1 steps in the `for` loop.
- The input parameter *k* is 0. Because *k* is an unsigned integer, the value of *k*-1 is 4294967295 (unsigned 0xFFFFFFFF) rather than -1. It iterates in the `for` loop many more times than expected.

There are too many risks of a crash in such a simple piece of code, so it is reasonable for the writer of this code to be rejected.

Tip Robustness is worthy of attention in interviews. Rejection is quite likely if there are many potential risks of crash in code.

Let's fix the listed problems one by one:

- If the pointer of the head node is `NULL`, the whole list is empty, so naturally the k^{th} node from the tail is also `NULL`.
- An `if` statement can be inserted to check whether it reaches a `NULL` address when it tries to move *k* steps on the list. When the total number of nodes in the list is less than *k*, it returns `NULL`.
- It counts from 1 here, and the first node from the tail is the tail node itself. It is an invalid input if *k* is 0 because it is meaningless to find the zeroth node from the tail. It returns `NULL` too in such a case.

The revised version of the code is shown in Listing 5-20.

Listing 5-20. *C++ Code to Get the kth Node from End (Version 2)*

```
   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;

}
```

Source Code:

```
    047_KthNodeFromEnd.cpp
```

Test Cases:

- Functional Cases (The k^{th} node is the head/tail of a list or inside a list)
- Cases for Robustness (The pointer to the list head is `NULL`; the number of a list is less than *k*; *k* is 0)

Reverse a List

Question 48 Please implement a function to reverse a list.

Lots of pointer operations are involved in solving problems related to linked lists. Interviewers know that many candidates are prone to making mistakes on pointer operations, so they like problems related to linked lists to qualify candidates' programming capabilities. Candidates should analyze and design carefully before they begin to write code. It is much better to write robust code with comprehensive analysis than to write code quickly with many bugs.

The direction of pointers should be reversed in order to reverse a linked list. Let's utilize figures to analyze visually the complex steps to reverse pointers. As shown in the list in Figure 5-7(a), node *h*, *i*, and *j* are three adjacent nodes. Supposing pointers to all nodes prior to *h* have been reversed after some operations and all next pointers are linked to their preceding nodes. The next step is to reverse the next pointer in node *i*. The status of the list is shown in Figure 5-7(b).

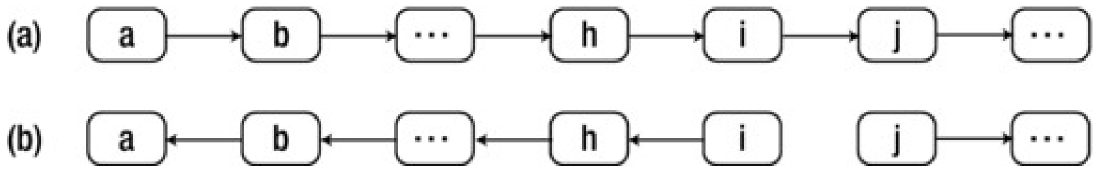

Figure 5-7. *A list gets broken when it is reversed. (a) A list. (b) All next pointers in nodes prior to the node* i *are reversed to reference their preceding nodes. The link between the nodes* i *and* j *gets disconnected.*

Notice that the next pointer in node *i* references its preceding node *h*, so the list is broken and node *j* is inaccessible (Figure 5-7(b)). Node *j* should be saved before the next pointer of node *i* is adjusted in order to prevent the list from becoming broken.

When the next pointer in node *i* is reversed, we need access to node *h* since the pointer is adjusted to reference node *h*. It also needs to access node *j*; otherwise, the list will be broken. Therefore, three pointers should be declared in our code that point to the current visited node, its preceding node, and its next node respectively.

Lastly, the head node of the reversed list should be returned. Obviously, the head in the reversed list should be the tail of the original list. Which pointer is the tail? It should be the node whose next pointer is `NULL`.

With comprehensive analysis above, we are ready to write code, which is shown in Listing 5-21.

Listing 5-21. *C++ Code to Reverse a List*

```
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;
}
```

The common issues remaining in many candidates' code are in three categories:

- The program crashes when the pointer of the head node is `NULL` or there is only one node in the list.
- The reversed list is broken.
- The returned node is the head node of the original list rather than the head node of the reversed list.

How do you make sure there are no problems remaining in code during interviews? A good practice is to have comprehensive test cases. After finishing writing code, candidates can test their own code with the prepared cases. The code can be handed to interviewers only after all unit tests are passed. Actually, interviewers have their own test cases to verify candidates' code. If a candidate's test covers all test cases prepared by his or her interviewer, it is highly possible to pass this round of interviews.

Source Code:

`048_ReverseList.cpp`

Test Cases:

- Functional Cases (The number of nodes in a list is even/odd; there is only one node in a list)
- Cases for Robustness (The pointer to the list head is `NULL`)

Question 49 How do you design an algorithm to reverse every *k* nodes in a list? A list is divided into several groups, and each group has *k* nodes except the last group, where the number of nodes may be less than *k*. Please reverse the nodes in each group and connect all groups together.

For example, when groups with three nodes are reversed in the list of Figure 5-8(a), it becomes the list in Figure 5-8(b).

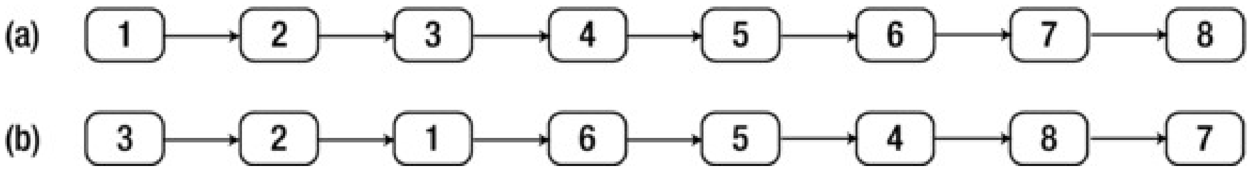

Figure 5-8. *Reverse every group with three nodes in a list with eight nodes.*

Three steps are necessary to reverse a list group by group. First, the process finds a group with *k* nodes (the number of nodes in the last group may be less than *k*). Then it reverses pointers to nodes inside the group. Finally, it connects reversed groups together. Therefore, the overall code structure looks like the function `Reverse`, as shown in Listing 5-22.

Listing 5-22. *C++ Code to Reverse Nodes in a List in Groups*

```
ListNode* Reverse(ListNode* pHead, unsigned int k) {
 if(pHead == NULL || k <= 1)
     return pHead;

 ListNode* pReversedHead = NULL;
 ListNode* pNode1 = pHead;
 ListNode* pPrev = NULL;
 while(pNode1 != NULL) {
     // find k nodes within a group
     ListNode* pNode2 = pNode1;
     ListNode* pNext = NULL;
     for(unsigned int i = 1; pNode2->m_pNext != NULL && i < k; ++i)
         pNode2 = pNode2->m_pNext;

     pNext = pNode2->m_pNext;

     // reverse nodes within a group
     ReverseGroup(pNode1, pNode2);

     // connect groups together
     if(pReversedHead == NULL)
         pReversedHead = pNode2;

     if(pPrev != NULL)
         pPrev->m_pNext = pNode2;
     pPrev = pNode1;
```

```
            pNode1 = pNext;

    }


    return pReversedHead;

}
```

The process required to reverse nodes inside a group is similar to the process discussed for the preceding problem, so it will not be analyzed step-by-step. The function `ReverseGroup` is used to reverse a group between two nodes, as shown in Listing 5-23.

Listing 5-23. *C++ Code to Reverse Nodes in a Group*

```
void ReverseGroup(ListNode* pNode1, ListNode* pNode2) {

    ListNode* pNode = pNode1;

    ListNode* pPrev = NULL;

    while(pNode != pNode2) {

        ListNode* pNext = pNode->m_pNext;

        pNode->m_pNext = pPrev;


        pPrev = pNode;

        pNode = pNext;

    }


    pNode->m_pNext = pPrev;

}
```

Source Code:

```
049_ReverseListInGroups.cpp
```

Test Cases:

- Functional Cases (The number of nodes in the last group of a list is *k* or less than *k*; the number of nodes in a list is less than or equal to *k*)
- Cases for Robustness (The pointer to the list head is `NULL`; *k* is 0 or negative)

Substructures in Trees

Question 50 Given two binary trees, please check whether one is a substructure of the other. For example, the tree *b* on the right side of Figure 5-9 is a substructure of the tree *a* on the left side.

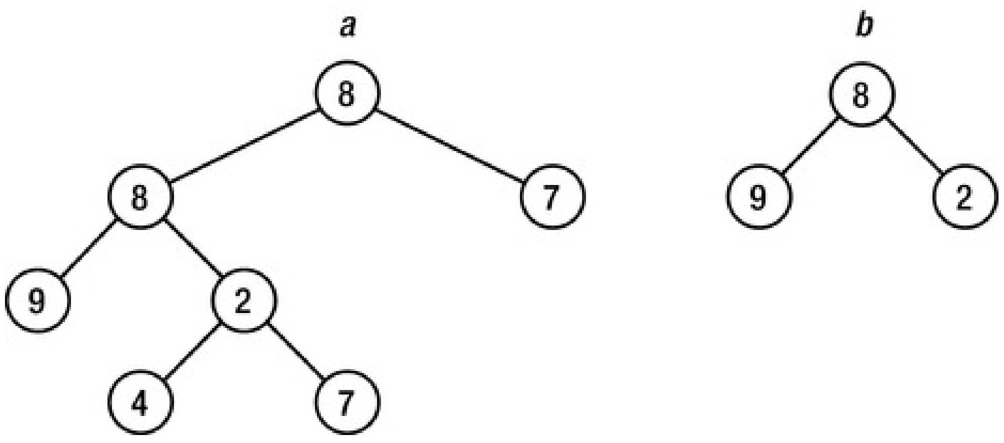

Figure 5-9. *Two binary trees of which tree* b *on the right side is a substructure of tree* a *on the left side.*

Similar to lists, pointer operations on trees are also quite complicated, so it is usually not easy to solve coding interview problems about trees. Candidates have to be very careful on pointer operations; otherwise, it is highly possible to leave crash risks.

Let's return to the problem itself. It can be solved in two steps. The first step is to find a node *r* in tree *a* whose value is the same as the value in the root node of tree *b*, and the second step is to check whether the subtree rooted at the node *r* has the same structure as tree *b*.

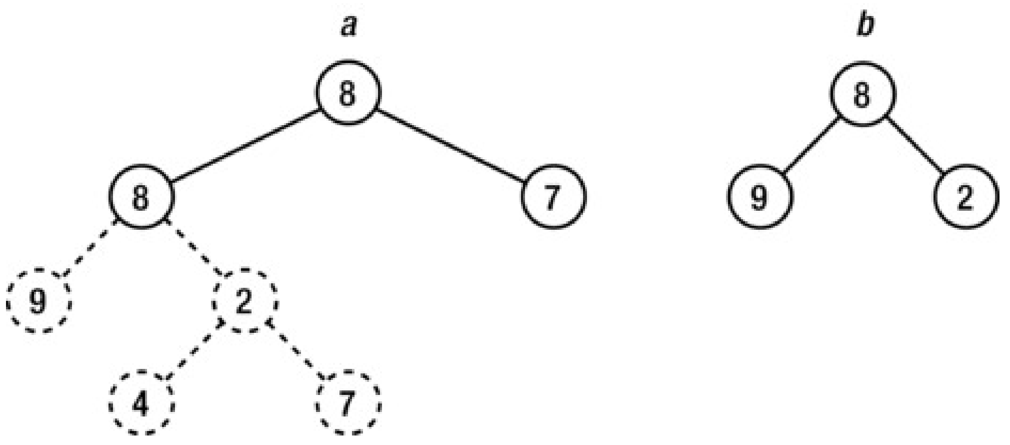

Figure 5-10. *Two root nodes in trees* a *and* b *have the same value, but their children are different.*

Take the two sample trees as an example. Our solution traverses tree *a* and finds a node with a value of 8 (the value in the root node of tree *b*). Since the value in the root of tree *a* is 8, the solution checks whether the subtree under the root node in tree *a* has the same structure as tree *b*. As shown in Figure 5-10, their structures are different.

It continues to traverse on tree *a*, and another node with value 8 is found in the second level. It checks the structure of the subtree again, and the left and right children have the same values as the children nodes in tree *b* (Figure 5-11). Therefore, a subtree in tree *a* with the same structure as tree *b* has been found, so tree *b* is a substructure of tree *a*.

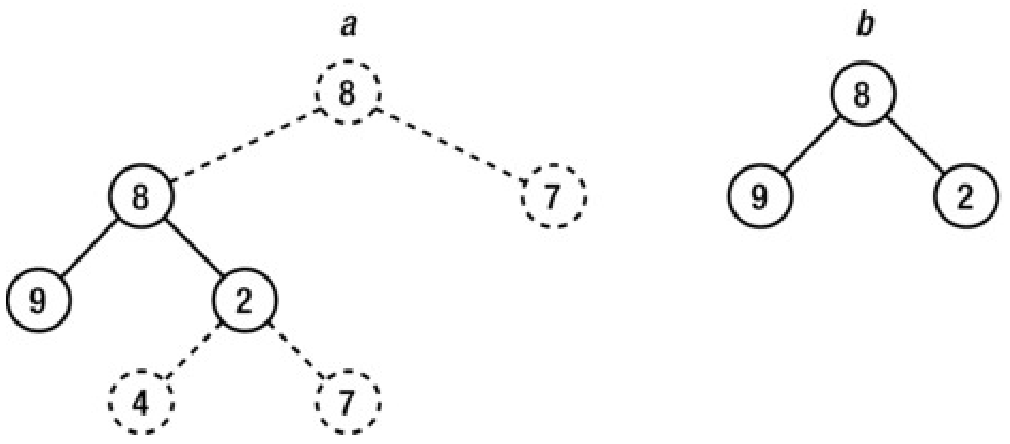

Figure 5-11. *The structure under the second node with value 8 in tree* a *is the same as the structure of tree* b.

The first step to find a node in tree *a* with the same value in the root node of tree *b* is actually a traversal process. There are different traversal algorithms available, and the following code is based on the pre-order traversal algorithm. The following function `HasSubtree` solves this problem recursively in Listing 5-24 because it is simpler to implement traversal algorithms with recursion.

Listing 5-24. *C++ Code Check Whether a Tree Is a Subtree of Another*

```
bool HasSubtree(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2) {
    bool result = false;

    if(pRoot1 != NULL && pRoot2 != NULL) {
        if(pRoot1->m_nValue == pRoot2->m_nValue)
            result = DoesTree1HaveTree2(pRoot1, pRoot2);
        if(!result)
            result = HasSubtree(pRoot1->m_pLeft, pRoot2);
        if(!result)
            result = HasSubtree(pRoot1->m_pRight, pRoot2);
    }

    return result;
}
```

In the function `HasSubtree`, it moves on to check whether a subtree in tree *a* has the same structure as tree *b* with the function `DoesTree1HaveTree2`, as shown in Listing 5-25.

Listing 5-25. *C++ Code Check Whether a Tree Is a Subtree of Another*

```
bool DoesTree1HaveTree2(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2) {
    if(pRoot2 == NULL)
        return true;
    if(pRoot1 == NULL)
        return false;

       if(pRoot1->m_nValue != pRoot2->m_nValue)
        return false;

    return DoesTree1HaveTree2(pRoot1->m_pLeft, pRoot2->m_pLeft) &&
        DoesTree1HaveTree2(pRoot1->m_pRight, pRoot2->m_pRight);
}
```

Notice that there are many places to check whether a pointer is `NULL`. Candidates have to be careful when writing code to traverse trees. They should ask themselves whether it is possible for a pointer to be `NULL` in every statement with pointers and how to handle it when the pointer is `NULL`. Programs are prone to crash when some `NULL` pointers are not handled appropriately.

Tip It is important to ask two questions when writing code with pointer operations: Is it possible for the pointer to be `NULL`? How do you handle it when the pointer is `NULL`?

Source Code:

`050_SubtreeInTree.cpp`

Test Cases:

- Functional Cases (A binary tree is/is not a subtree of another)
- Cases for Robustness (The pointers to the head of one or two binary trees are `NULL`; some special binary trees where nodes do not have left/right subtrees)