

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.

Figures to Visualize Problems

Figures are helpful tools to analyze and solve problems. When interview problems are complex, it is a good practice for candidates to visualize them with figures, which may help uncover hidden rules. Many candidates get inspiration when drawing figures or say “Gotcha” while staring at figures.

Figures are extremely useful in analyzing problems about data structures such as binary trees, 2D matrices, and lists. Sometimes it is difficult to find a solution after meditating for a long time over the problem, but with a few figures you can get the hang of it. Take the problem “Mirror of Binary Trees” (Question 51) as an example. The operations to get a mirrored image of a binary tree are not simple. However, if we draw some figures using binary trees and their mirrored images, we may find that it is only necessary to swap the left and right children nodes of each node while traversing.

Figures are also quite helpful tools to facilitate communications with interviewers. It can be difficult to explain complicated solutions only with spoken words. However, as we know, a picture is worth a thousand words. It is much easier for interviewers to understand the candidates’ ideas if candidates draw some figures while explaining their solutions. The habit of drawing intelligible figures while explaining solutions is a demonstration of strong communication skills during interviews.

Mirror of Binary Trees

Question 51 Given a binary tree, how do you get its mirrored tree?

The mirrored tree is a new concept for many candidates. If you cannot find a solution in a short time, you may try to draw a binary tree and its mirrored image. The tree on the right of Figure 6-1 is the mirrored image of the tree on the left.

Let’s try to figure out the steps needed to get a mirrored tree by scrutinizing these two trees. Their root nodes are the same, but their left and right children are swapped. Therefore, two nodes under the root are swapped and the result is shown as the second tree in Figure 6-2.

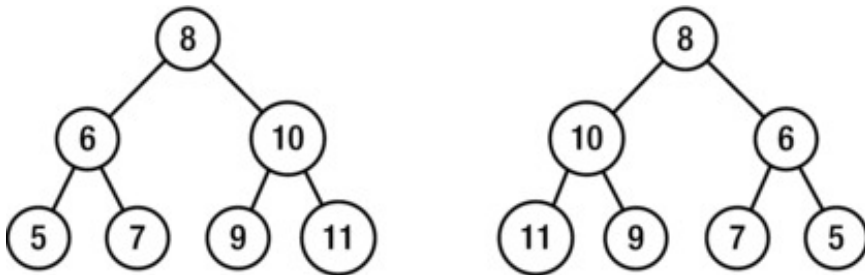


Figure 6-1. Two binary trees of which one is the mirrored tree of the other

After two nodes under the root have been swapped, notice that the order of children nodes under these two nodes is different from the mirrored target tree. Therefore, it continues to swap children nodes and gets the third and fourth trees in Figure 6-2. The fourth tree looks the same as the mirrored image of the original tree.

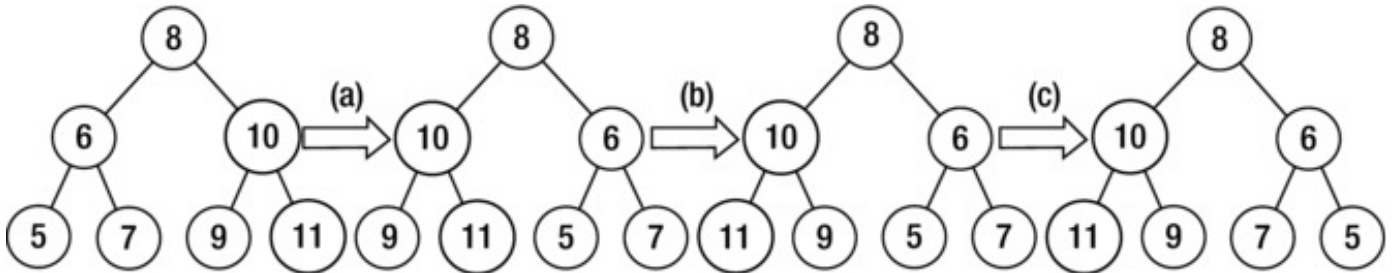


Figure 6-2. The process to get a mirrored tree. (a) Swap the left and right subtrees of the root node. (b) Swap the left and right subtrees of the node with value 10. (c) Swap the left and right subtrees of the node with value 6.

The process to get a mirrored tree can be summarized in a couple of sentences. Scan a binary tree with the pre-order traversal algorithm. When a node is visited, its children nodes are swapped.

This solution can be implemented based on recursion, as shown in Listing 6-1.

Listing 6-1. C++ Code to Get a Mirrored Image of a Binary Tree

```
void MirrorRecursively(BinaryTreeNode *pNode) {
    if(pNode == NULL)
        return;

    if(pNode->m_pLeft == NULL && pNode->m_pRight == NULL)
        return;

    BinaryTreeNode *pTemp = pNode->m_pLeft;
    pNode->m_pLeft = pNode->m_pRight;
    pNode->m_pRight = pTemp;

    if(pNode->m_pLeft)
        MirrorRecursively(pNode->m_pLeft);
}
```

```

if(pNode->m_pRight)
    MirrorRecursively(pNode->m_pRight);
}

```

Source Code:

051_MirrorOfBinaryTree.cpp

Test Cases:

- Functional Cases (Normal binary trees)
- Cases for Robustness (The pointer to the head of a binary tree is `NULL` ; a binary tree with only one node; some special binary trees where nodes do not have left/right subtrees)

Question 52 Please implement a function to verify whether a binary tree is symmetrical. A tree is symmetrical if its mirrored image looks the same as the tree itself. There are three binary trees in [Figure 6-3](#). The first tree is symmetrical, but the second and the third are not.

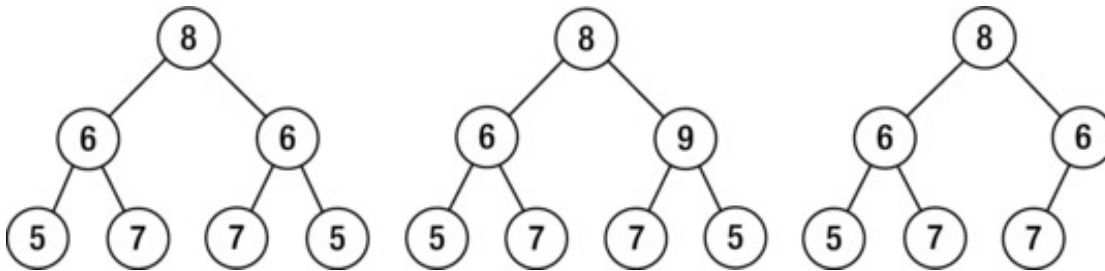


Figure 6-3. Three binary trees. The first tree is symmetrical, but the other two are not.

As we know, there are three common depth-first traversals, which are pre-order, in-order, and post-order traversals. Left children are visited prior to right children with these algorithms. What if we define a mirrored traversal algorithm that visits right children before left children? For example, it visits the root node first, then the left children, and finally the right children with the pre-order traversal algorithm. If the mirrored pre-order traversal algorithm is applied, it visits the root node first, then the right children, and finally the left children.

If the first tree in [Figure 6-3](#) is scanned with the pre-order traversal algorithm, the traversal sequence is {8, 6, 5, 7, 6, 7, 5}. If the mirrored pre-order traversal is applied, its traversal sequence is {8, 6, 5, 7, 6, 7, 5}. Notice that these two sequences are identical to each other.

The traversal sequence of the second tree is {8, 6, 5, 7, 9, 7, 5} if the pre-order traversal algorithm is applied, and the sequence is {8, 9, 5, 7, 6, 7, 5} with the mirrored pre-order traversal. The second and fifth steps are different. Similarly, the traversal sequences are {8, 6, 5, 7, 6, 7} and {8, 6, 7, 6, 7, 5} for the third tree with the normal and mirrored pre-order traversals. They differ from each other starting with the third step.

We have found that the pre-order traversal sequence and mirrored pre-order traversal sequence are the same when a binary tree is symmetrical; otherwise, these two sequences are different. Therefore, we can verify whether a tree is symmetrical with the code shown in [Listing 6-2](#).

Listing 6-2. C++ Code to Verify Symmetrical Trees

```

bool isSymmetrical(BinaryTreeNode* pRoot) {
    return isSymmetrical(pRoot, pRoot);
}

bool isSymmetrical(BinaryTreeNode* pRoot1, BinaryTreeNode* pRoot2) {
    if(pRoot1 == NULL && pRoot2 == NULL)
        return true;

    if(pRoot1 == NULL || pRoot2 == NULL)
        return false;

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

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

```

Source Code:

052_SymmetricalBinaryTrees.cpp

Test Cases:

- Functional Cases (Ordinary binary trees are/are not symmetrical)
- Cases for Robustness (The pointer to the head of a binary tree is `NULL` ; a binary tree with only one node; some special

binary trees where nodes do not have left/right subtrees)

Print Matrix in Spiral Order

Question 53 Please print a matrix in spiral order, clockwise from outer rings to inner rings. For example, the matrix below is printed in the sequence of 1, 2, 3, 4, 8, 12, 16, 15, 14, 13, 9, 5, 6, 7, 11, 10.

```
1 2 3 4
5 6 7 8
9 10 11 12
13 14 15 16
```

It looks like a simple problem because it is not about any complex data structures or advanced algorithms. However, the source code to solve this problem contains many loops with lots of boundary values. Many candidates find themselves in a pickle if they begin to write code before they get clear ideas of all the issues involved.

Figures are helpful tools to analyze problems. Since it is required to print to a matrix from outer rings to inner ones, a matrix is viewed as a set of concentric rings.

Figure 6-4 shows a ring in a square matrix. A matrix can be printed in a `for` or `while` loop starting with outer rings and moving to the interior in each iteration.

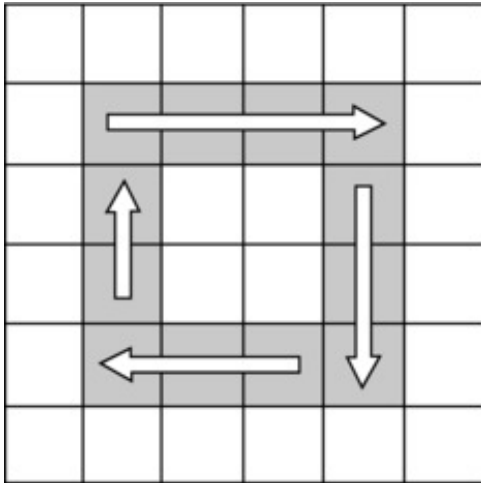


Figure 6-4. A matrix is composed of a set of rings.

Let's analyze when to end the iteration. Suppose there are r rows and c columns in a matrix. Notice the row index and column index are always identical in the beginning element in each ring at the top left corner. The index of the beginning element in the i^{th} ring is denoted as (i, i) . The statements $c > i \times 2$ and $r > i \times 2$ are always true for all rings in a matrix. Therefore, a matrix can be printed iteratively with the code shown in Listing 6-3.

Listing 6-3. Java Code to Print a Matrix

```
void printMatrixClockwise(int numbers[][]){
    int rows = numbers.length;
    int columns = numbers[0].length;
    int start = 0;

    while(columns > start * 2 && rows > start * 2){
        printRing(numbers, start);

        ++start;
    }
}
```

Let's move on to implement the method `printRing` to print a ring in a matrix. As shown in Figure 6-4, a ring can be printed in four steps. It prints a row from left to right in the first step, a column in top down order in the second step, then another row from right to left in the third step, and finally a column bottom up.

There are many corner cases worthy of attention. The innermost ring in a matrix might only have a column, a row, or even an element. Some corner cases are included in Figure 6-5, where it only needs three steps, two steps, or even one step to print the innermost ring.

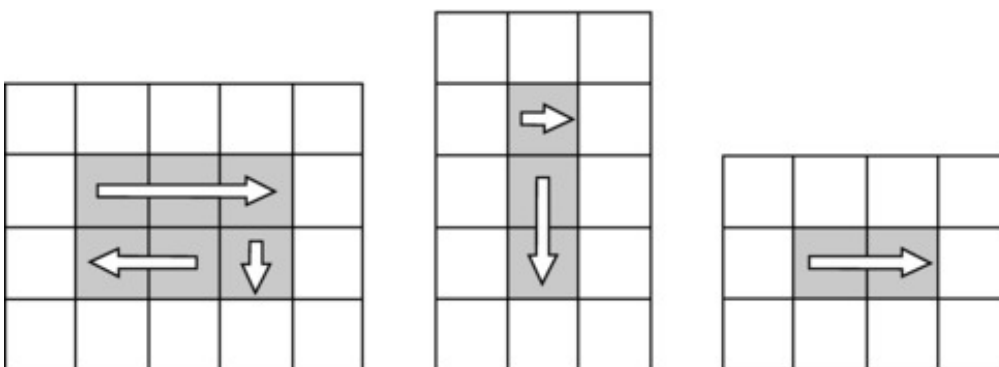


Figure 6-5. It may take three, two, or even one step to print the last ring in a matrix.

We have to analyze the prerequisites for each step. The first step is always necessary since there is at least one element in a ring. The second step is not needed if there is only one row remaining in the last ring. Similarly, the third step is needed when there are two rows and two columns at least in a ring, and the fourth step is needed when there are three rows and two columns. Therefore, the method `printRing` can be implemented as shown in [Listing 6-4](#).

Listing 6-4. Java Code to Print a Ring in a Matrix

```
void printRing(int numbers[][], int start) {

    int rows = numbers.length;

    int columns = numbers[0].length;

    int endX = columns - 1 - start;

    int endY = rows - 1 - start;

    // Print a row from left to right

    for(int i = start; i <= endX; ++i) {

        int number = numbers[start][i];

        printNumber(number);

    }

    // print a column top down

    if(start < endY) {

        for(int i = start + 1; i <= endY; ++i) {

            int number = numbers[i][endX];

            printNumber(number);

        }

    }

    // print a row from right to left

    if(start < endX && start < endY) {

        for(int i = endX - 1; i >= start; --i) {

            int number = numbers[endY][i];

            printNumber(number);

        }

    }

    // print a column bottom up

    if(start < endX && start < endY - 1) {

        for(int i = endY - 1; i >= start + 1; --i) {

            int number = numbers[i][start];

            printNumber(number);

        }

    }

}
```


Source Code:

053_PrintMatrix.java

Test Cases:

- Functional Cases (A matrix with multiple rows and columns)
- Boundary Cases (A matrix with only a row, a column, or even an element)

Clone Complex Lists

 **Question 54** Please implement a function to clone a complex list. Every node in a complex list has a link `m_pSibling` to an arbitrary node in the list besides the link `m_pNext` to the next node.

For example, there is a complex list in [Figure 6-6](#) with five nodes. The dashed arrows are `m_pSibling` links, and the normal arrows are `m_pNext` links.

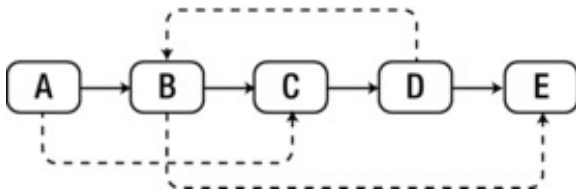


Figure 6-6. A complex list with five nodes. Normal arrows are links to the next nodes, and dashed arrows are links to arbitrary nodes. Pointers to NULL are not drawn for simplicity.

It is not difficult to get the brute-force solution in two steps. In the first step, clone every node in the original list and link all cloned nodes with `m_pNext` pointers. In the second step, set `m_pSibling` links on the cloned list. Let's suppose `m_pSibling` of node N points to node S in the original list, and the cloned node of N is N' . If the distance between the head node and node S in the original list is s , the distance between the head node and node S' , referenced by `m_pSibling` of N' in the cloned list, should also be s .

If there are n nodes in a list, this solution has to move $O(n)$ steps beginning from the head node to locate the target of a `m_pSibling` link. Therefore, the overall time complexity of this solution is $O(n^2)$.

Notice that most of the time is spent on locating targets of `m_pSibling` links. We can improve time efficiency with another solution. The new solution also clones a list in two steps. In the first step, it creates a node N' and clones data from every node N in the original list and links all cloned nodes together with `m_pNext` pointers. Additionally, all node pairs $\langle N, N' \rangle$ are saved in a hash table where N is a key and N' is a value. The second step is also to set `m_pSibling` links on the cloned list. Suppose `m_pSibling` of node N points to node S in the original list and their corresponding nodes are N' and S' in the cloned list. This solution can locate S' in $O(1)$ time in the hash table with node S .

The second solution sacrifices space efficiency for time efficiency. It needs a hash table with $O(n)$ size if there are n nodes in a list, and it reduces the time complexity to $O(n)$ from $O(n^2)$.

Let's explore another solution with $O(n)$ time efficiency and without auxiliary space consumption. Similar to before, this solution also creates new nodes and clones the data of the original nodes. However, the cloned node N' is linked next to the original node N . The list in Figure 6-6 becomes longer after this step, as shown in Figure 6-7.

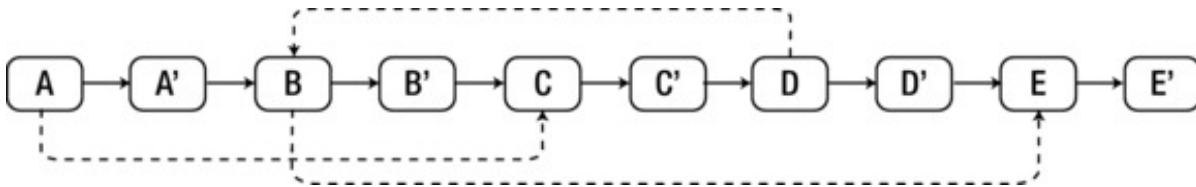


Figure 6-7. The first step to clone a complex list. A node N' is created in order to clone a node N , and it is linked as the next node of N .

The source code for the first step to clone a complex list is shown in Listing 6-5.

Listing 6-5. C++ Code of the First Step to Clone a Complex List

```
void CloneNodes(ComplexListNode* pHead) {
    ComplexListNode* pNode = pHead;
    while(pNode != NULL) {
        ComplexListNode* pCloned = new ComplexListNode();
        pCloned->m_nValue = pNode->m_nValue;
        pCloned->m_pNext = pNode->m_pNext;
        pCloned->m_pSibling = NULL;

        pNode->m_pNext = pCloned;
        pNode = pCloned->m_pNext;
    }
}
```

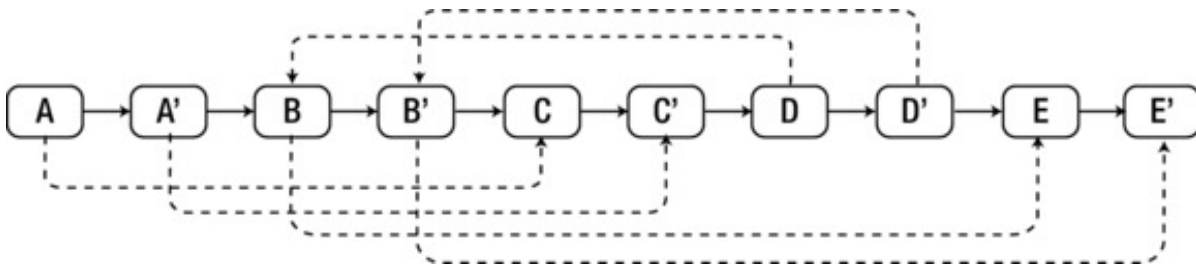


Figure 6-8. The second step to clone a complex list. If the `m_pSibling` of a node N points to S , the `m_pSibling` of the cloned node N' points to S' , which is the next node of S .

The second step is to set `m_pSibling` for each cloned node. Suppose that originally the `m_pSibling` in node N points to node S . Node N' , the cloned node of N , is the next node of N , and node S' is also the next node of S . Therefore, it locates the `m_pSibling` target of node N' in $O(1)$ time. The list after setting all `m_pSibling` links is shown in Figure 6-8.

The code for the second step is shown in Listing 6-6.

Listing 6-6. C++ Code of the Second Step to Clone a Complex List

```
void ConnectSiblingNodes(ComplexListNode* pHead) {
```

```

ComplexListNode* pNode = pHead;

while(pNode != NULL) {

    ComplexListNode* pCloned = pNode->m_pNext;

    if(pNode->m_pSibling != NULL) {

        pCloned->m_pSibling = pNode->m_pSibling->m_pNext;

    }

    pNode = pCloned->m_pNext;

}
}

```

It splits the long list into two lists in the third step. The nodes with odd indexes are linked together with `m_pNext` pointers, which reform the original list. The nodes with even indexes are linked together with `m_pNext` pointers too, and they compose the cloned list, as shown in [Figure 6-9](#).

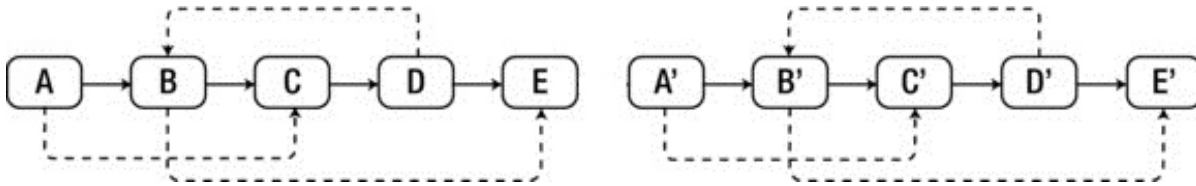


Figure 6-9. The third step to clone a complex list, which splits the list in [Figure 6-8](#) into two lists. The nodes with odd indexes (1st, 3rd, 5th, and so on) compose the original list, while others compose the cloned list.

The code for the third step is shown in [Listing 6-7](#).

Listing 6-7. C++ Code of the Third Step to Clone a Complex List

```

ComplexListNode* ReconnectNodes(ComplexListNode* pHead) {

    ComplexListNode* pNode = pHead;

    ComplexListNode* pClonedHead = NULL;

    ComplexListNode* 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;

}

```

It is the whole process to clone a complex list when the three steps above are combined together, as shown in [Listing 6-8](#).

Listing 6-8. C++ Code of the Third Step to Clone a Complex List

```

ComplexListNode* Clone(ComplexListNode* pHead) {

    CloneNodes(pHead);

    ConnectSiblingNodes(pHead);

    return ReconnectNodes(pHead);

}

```

Source Code:

054_CloneComplexList.cpp

Test Cases:

- Functional Cases (Lists with multiple nodes and `m_pSibling` links)
- Boundary Cases (There is only one node in a list; there are loops with `m_pSibling` links, or some of the `m_pSibling` links are connected to their owner nodes)

- Cases for Robustness (the pointer to the head node of a list is `NULL`)