CHAPTER 6

Approaches to Solutions

There are often tough situations where candidates are asked difficult interview questions. This chapter provides three strategies to solve complex problems:

- Figures may be helpful to visualize data structures problems such as lists and trees.
- Step-by-step analysis with examples may uncover hidden rules.
- The divide-and-conquer approach breaks down a problem into manageable pieces and solves them recursively.

Each strategy is illustrated with several sample coding interview questions.

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.

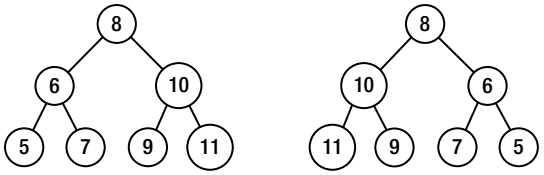

Figure 6-1. *Two binary trees of which one is the mirrored tree of the other*

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.

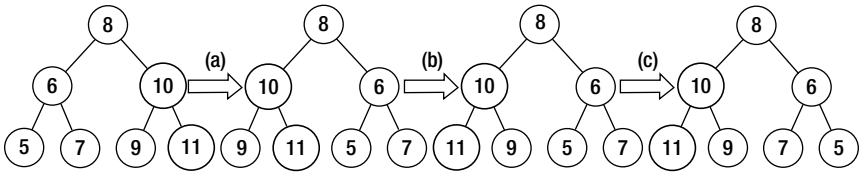

Figure 6-2. *The process to get a mirrored tree. (a) Swap the left and right subtrees of the root node. (b) Swap the left and right subtrees of the node with value 10. (c) Swap the left and right subtrees of the node with value 6.*

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.

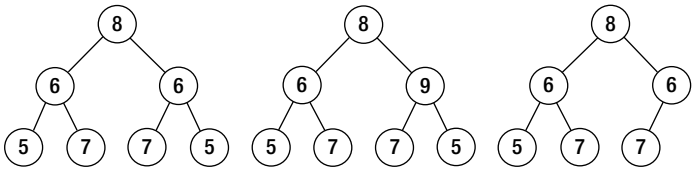

Figure 6-3. *Three binary trees. The first tree is symmetrical, but the other two are not.*

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.

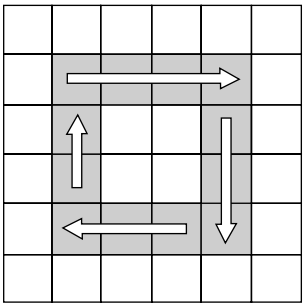

Figure 6-4. *A matrix is composed of a set of rings.*

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.

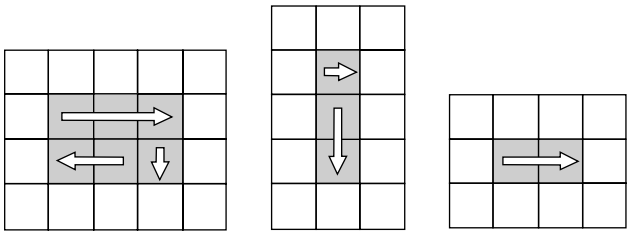

Figure 6-5. *It may take three, two, or even one step to print the last ring in a matrix.*

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.

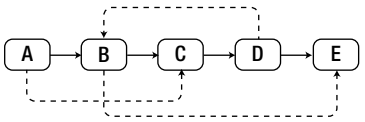

Figure 6-6. *A complex list with five nodes. Normal arrows are links to the next nodes, and dashed arrows are links to arbitrary nodes. Pointers to NULL are not drawn for simplicity.*

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 $<N, N'>$ 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.

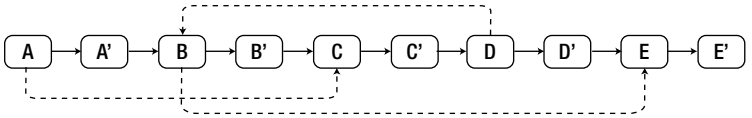

Figure 6-7. *The first step to clone a complex list. A node* N′ *is created in order to clone a node* N*, and it is linked as the next node of* N.

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;
    }
}
```

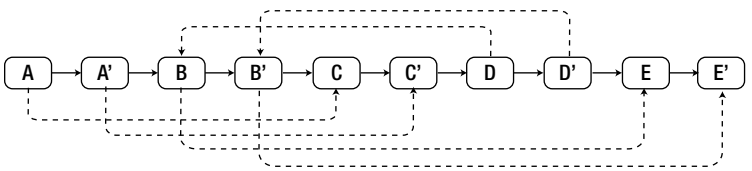

Figure 6-8. *The second step to clone a complex list. If the m_pSlibling of a node* N *points to* S*, the m_pSlibling of the cloned node* N′ *points to* S′*, which is the next node of* S.

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.

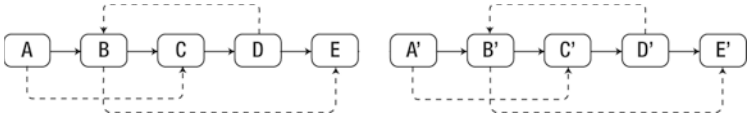

Figure 6-9. *The third step to clone a complex list, which splits the list in Figure 6-8 into two lists. The nodes with odd indexes (1st, 3rd, 5th, and so on) compose the original list, while others compose the cloned list.*

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`)

Examples to Simplify Problems

Similar to figures, examples are also helpful tools to analyze and solve complex problems. It is quite common for candidates to find solutions when they utilize examples to simulate complicated processes step-by-step. For instance, many candidates cannot find rules to push and pop quickly for the interview question "Push and Pop Sequences of Stacks" (Question 56). When meeting such a problem, candidates may take one or two sequences as examples to simulate the pushing and popping operations. It is much easier to uncover the hidden rules with step-by-step analysis.

Examples can help candidates to communicate with interviewers. Abstract algorithms are usually difficult to explain only with oral words. In such situations, candidates may tell interviewers how their algorithms handle some sample inputs in a stepwise manner. Take the interview problem "Print Binary Trees in Zigzag Order" (Question 59) as an example. Candidates may take some sample binary trees with multiple levels as samples to illustrate why they need two stacks.

Examples are great tools to assure code quality during interviews. It is a good habit for candidates to review their code carefully and make sure that there are no bugs before they hand code to interviewers. How do you check for bugs? The answer is test cases. The examples utilized to analyze problems are test cases. Candidates can simulate execution in their minds. If operation results at every step on all examples are the same as expected, the code quality is assured.

Stack with Min Function

An intuitive solution for this problem might be that it sorts all numbers in the stack when it pushes a new element and keeps the minimum number on the top of stack. In this way, we can get the minimum number in O(1) time. However, it cannot be guaranteed that the last number pushed in to the container will be the first one to be popped off, so the data container is no longer a stack.

■ **Question 55** Define a stack in which we can get its minimum number with a function `min`. The time complexity of `min`, `push`, and `pop` on such stacks are all O(1).

A new field variable may be added in a stack to keep the minimum number. When a new number that is less than the minimum number is pushed, the minimum gets updated. It sounds good. However, how do you get the next minimum when the current minimum is popped? This naïve solution does not work either.

Let's explore other alternatives.

With an Auxiliary Stack

It is not enough just to keep a field variable as the minimum number. When the minimum is popped, the solution should be able to find the next minimum. Therefore, it is necessary to restore the next minimum number after the current minimum one is popped off.

How about storing each minimum number (the lesser value of the number to be pushed and the minimum number at that time) into an auxiliary stack? Let's analyze the process needed to push and pop numbers via some examples (Table 6-1).

At first, the solution pushes 3 into both data stack and auxiliary stack. The second number to be pushed into the data stack is the number 4. It pushes 3 again into the auxiliary stack because the number 4 is greater than 3. Third, it continues pushing 2 into the data stack. It updates the minimum number to 2 and pushes 2 into the auxiliary stack since 2 is less than the previous minimum number 3. The fourth step to push 1 is similar. The minimum number is updated to 1, and it is pushed into the auxiliary stack. Notice that the top of the auxiliary stack is always the minimum number in each step.

Table 6-1. *The Status of the Data Stack, Auxiliary Stack, and Minimum Number When It Pushes the Numbers 3, 4, 2, 1, Pops Twice, and Then Pushes 0*

Step	Operation	Data Stack	Auxiliary Stack	Minimum
1	Push 3	3	3	3
2	Push 4	3, 4	3, 3	3
3	Push 2	3, 4, 2	3, 3, 2	2
4	Push 1	3, 4, 2, 1	3, 3, 2, 1	1
5	Pop	3, 4, 2	3, 3, 2	2
6	Pop	3, 4	3, 3	3
7	Push 0	3, 4, 0	3, 3, 0	0

Whenever a number is popped from the data stack, a number is also popped from the auxiliary stack. If the minimum number is popped, the next minimum number should also be on the top of the auxiliary stack. In the fifth step, it pops 1 from the data stack, and it also pops the number on the top of the auxiliary, which is 1. The next minimum number, 2, is now on the top of the auxiliary stack. If it

continues popping from both the data and auxiliary stacks, there are only two numbers, 3 and 4, left in the data stack. The minimum number, 3, is indeed on the top of the auxiliary stack. If a new number, 0, is pushed into the data stack, as well as the auxiliary stack, the number on top of the auxiliary stack is the minimum 0.

This step-by-step analysis demonstrates that our solution is correct. Now it is time to implement the required stack, which can be defined as the code found in Listing 6-8.

Listing 6-8. *C++ Code for a Stack with Function Min (Version 1)*

```
template <typename T> class StackWithMin {
public:
    StackWithMin(void) {}
    virtual ~StackWithMin(void) {}

    T& top(void);

    void push(const T& value);
    void pop(void);

    const T& min(void) const;

private:
    std::stack<T>   m_data;     // data stack, to store numbers
    std::stack<T>   m_min;      // auxiliary stack, to store minimum numbers
};

template <typename T> void StackWithMin<T>::push(const T& value) {
    m_data.push(value);

    if(m_min.size() == 0 || value < m_min.top())
        m_min.push(value);
    else
        m_min.push(m_min.top());
}

template <typename T> void StackWithMin<T>::pop() {
    assert(m_data.size() > 0 && m_min.size() > 0);

    m_data.pop();
    m_min.pop();
}

template <typename T> const T& StackWithMin<T>::min() const {
    assert(m_data.size() > 0 && m_min.size() > 0);

    return m_min.top();
}

template <typename T> T& StackWithMin<T>::top() {
    return m_data.top();
}
```

The length of the auxiliary stack is n if it pushes n numbers into the data stack. Therefore, it takes $O(n)$ auxiliary memory for this solution.

Without an Auxiliary Stack

The second solution is trickier without an auxiliary stack. It requires arithmetic before pushing rather than always pushing numbers into the data stack directly.

Suppose that the solution is going to push a number *value* into a stack with minimum number *min*. If *value* is greater than or equal to *min*, it is pushed directly into the data stack. If it is less than *min*, it pushes $2\times value-min$ into the stack and updates *min* as *value* since a new minimum number is pushed.

How about pop? It pops directly if the number at the top of the data stack (denoted as *top*) is greater than or equal to *min*. Otherwise, the number on the top is not the real pushed number. The real pushed number is *min*. After the current minimum number is popped, it restores the previous minimum number, which is $2\times min-top$.

Now let's demonstrate the correctness of this solution. When *value* is greater than or equal to *min*, it is pushed into the data stack directly without updating *min*. Therefore, when the number on the top of the stack is greater than or equal to *min*, it is popped off directly without updating *min*.

However, if *value* is less than *min*, it pushes $2\times value-min$. Notice that $2\times value-min=value+(value-min)$, and $value+(value-min)$ should be less than *value* when *value* is less than *min*. Then the solution updates current *min* as *value*, so the new top of the data stack (*top*) is less than the current *min*. Therefore, when the number on the top of the data stack is less than *min*, the real pushed number value is stored in *min*. After the number on top of the stack is popped off, it has to restore the previous minimum number (denoted as *min*'). Since $top=2\times value-min'$ and *value* is the current *min*, the previous minimum number is restored with $min'=2\times min-top$.

It sounds great. We now feel confident to write code with the correctness demonstrated. The code in Listing 6-10 is the sample code.

Listing 6-10. *C++ Code for a Stack with Function Min (Version 2)*

```
template <typename T> class StackWithMin {
public:
    StackWithMin(void) {}
    virtual ~StackWithMin(void) {}

    T& top(void);

    void push(const T& value);
    void pop(void);

    const T& min(void) const;

private:
    std::stack<T>   m_data;     // data stack, to store numbers
    T               m_min;      // minimum number
};

template <typename T> void StackWithMin<T>::push(const T& value) {
    if(m_data.size() == 0) {
        m_data.push(value);
        m_min = value;
```

```
    }
    else if(value >= m_min) {
        m_data.push(value);
    }
    else {
        m_data.push(2 * value - m_min);
        m_min = value;
    }
}

template <typename T> void StackWithMin<T>::pop() {
    assert(m_data.size() > 0);

    if(m_data.top() < m_min)
        m_min = 2 * m_min - m_data.top();

    m_data.pop();
}

template <typename T> const T& StackWithMin<T>::min() const {
    assert(m_data.size() > 0);

    return m_min;
}

template <typename T> T& StackWithMin<T>::top() {
    T top = m_data.top();
    if(top < m_min)
        top = m_min;

    return top;
}
```

In this solution, it is not necessary to have an auxiliary stack with O(*n*) elements, so it is more efficient from the perspective of memory utilization than the first solution.

Source Code:

`055_MinInStack.cpp`

Test Cases:

- The number to be pushed is less/greater than or equal to the current minimum number in a stack
- The number to be popped is/is not the current minimum number in a stack

Push and Pop Sequence of Stacks

■ **Question 56** You are given two integer arrays, one of which is a sequence of numbers pushed into a stack (supposing all numbers are unique). Please check whether the other array is a corresponding sequence popped from the stack.

For example, if the pushing sequence is {1, 2, 3, 4, 5}, the sequence {4, 5, 3, 2, 1} is a corresponding popping sequence, but {4, 3, 5, 1, 2} is not.

An auxiliary stack is utilized to stimulate pushing and popping operations, and the order of operations is defined by the two pushing and popping sequences. Let's simulate pushing and popping operations step-by-step in the sample sequences.

The first number to be popped off is 4 in the popping sequence {4, 5, 3, 2, 1}. The number 4 should be pushed into the auxiliary stack before it is popped off. The pushing order is defined by the sequence {1, 2, 3, 4, 5}, so numbers 1, 2 and 3 are pushed into the stack before the number 4. Four numbers (1, 2, 3, and 4) are in the stack at this time, and the number 4 is on top.

After the number 4 is popped off, three numbers (1, 2, and 3) are left. The next number to be popped is 5, but 5 has not been pushed into the stack yet. Therefore, numbers left in the pushing sequence are pushed into the stack. When the number 5 is pushed, it can be popped, and three numbers (1, 2, and 3) are left in the stack.

The next numbers to be popped off are 3, 2, and 1, and they are on the top of the stack before popping operations, so there are no problems popping them off one by one.

The steps to push and pop numbers according the pushing sequence {1, 2, 3, 4, 5} and popping sequence {4, 5, 3, 2, 1} are summarized in Table 6-2.

Table 6-2. *The Process Used to Push and Pop Elements According to the Pushing Sequence {1, 2, 3, 4, 5} and Popping Sequence {4, 5, 3, 2, 1}*

Step	Operation	Stack	Popped	Step	Operation	Stack	Popped
1	Push 1	1		6	Push 5	1, 2, 3, 5	
2	Push 2	1, 2		7	Pop	1, 2, 3	5
3	Push 3	1, 2, 3		8	Pop	1, 2	3
4	Push 4	1, 2, 3, 4		9	Pop	1	2
5	Pop	1, 2, 3	4	10	Pop		1

Let's take the popping sequence {4, 3, 5, 1, 2} as another example. Similar to before, numbers prior to 4 in the pushing sequence should be pushed into the stack because the number 4 is the first number to be popped off. The next number to be popped is 3, and it is on the top of the stack, so it is popped directly.

At this time there are two numbers, 1 and 2, in the stack. The number 5 is not in the stack, which is the next number to be popped. Numbers remaining in the pushing sequence are pushed into the stack. The number 5 can be popped after it is pushed into the stack.

The next number to be popped off is 1, but it is not on the top of the stack. The number on the top of the stack is 2. Additionally, there are no numbers remaining in the pushing sequence, so no more numbers can be pushed into the stack. We cannot continue to pop numbers, as shown in Table 6-3.

Table 6-3. *The Process to Push and Pop Elements According to the Pushing Sequence {1, 2, 3, 4, 5} and Popping Sequence {4, 3, 5, 1, 2}*

Step	Operation	Stack	Popped	Step	Operation	Stack	Popped
1	Push 1	1		6	Pop	1, 2	3
2	Push 2	1, 2		7	Push 5	1, 2, 5	
3	Push 3	1, 2, 3		8	Pop	1, 2	5
4	Push 4	1, 2, 3, 4		The next number to be popped is 1, but 1 is not on the top of the stack.			
5	Pop	1, 2, 3	4				

Let's summarize the process analyzed in detail. When the next number to be popped is on top of the stack, it is popped off directly. If it is not, more numbers are pushed into the stack. If the next number to be popped is not found when all numbers have been pushed, the sequence cannot be a popping sequence.

We feel confident enough to write the code after we have clear ideas about the pushing and popping process. The sample code is shown in Listing 6-11.

Listing 6-11. *C++ Code for Pushing and Popping Sequences of Stacks*

```
bool IsPopOrder(const int* pPush, const int* pPop, int nLength) {
    bool bPossible = false;

    if(pPush != NULL && pPop != NULL && nLength > 0) {
        const int* pNextPush = pPush;
        const int* pNextPop = pPop;
        std::stack<int> stackData;

        while(pNextPop - pPop < nLength) {
            // Push some numbers when the number to be popped is not
            // is not on the top of the stack
            while(stackData.empty() || stackData.top() != *pNextPop) {
                // Break when all numbers have been pushed
                if(pNextPush - pPush == nLength)
                    break;

                stackData.push(*pNextPush);
                pNextPush ++;
```

```
            }

            if(stackData.top() != *pNextPop)
                break;

            stackData.pop();
            pNextPop ++;
        }

        if(stackData.empty() && pNextPop - pPop == nLength)
            bPossible = true;
    }

    return bPossible;
}
```

Source Code:

`056_StackPushPopOrder.cpp`

Test Cases:

- Functional Cases (The pushing and popping sequences contain one or more numbers; an array is/is not the popping sequence corresponding to the pushing sequence in the other array)
- Cases for Robustness (One or two pointers to the arrays are `NULL`)

Print Binary Trees Level by Level

■ **Question 57** Please print a binary tree from its top level to bottom level, and print nodes at the same level from left to right.

For example, the binary tree in Figure 6-10 is printed in the sequence of 8, 6, 10, 5, 7, 9, 11.

This problem examines candidates' understanding of tree traversal algorithms, but the traversal here is not the common pre-order, in-order, or post-order traversals. If you are not familiar with it, you may analyze the printing process with some examples during the interview. Let's take the binary tree in Figure 6-10 as an example.

Since we begin to print from the top level of the tree in Figure 6-10, we can start our analysis from its root node. First, the value of the root node is printed, which is 8. You can store its children nodes with values 6 and 10 in a data container in order to print them after the root is printed. There are two nodes in the container at this time.

Second, node 6 is retrieved from the container prior to node 10 since nodes 6 and 10 are at the same level and we need to print them from left to right. Nodes 5 and 7 are stored to the container after node 6 is printed. There are three nodes in the container now, which are nodes 10, 5, and 7.

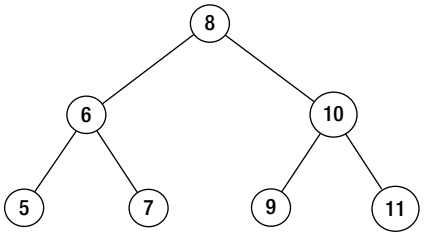

Figure 6-10. *A sample binary tree with three levels*

Third, we retrieve node 10 from the container. Notice that node 10 is stored in the container before nodes 5 and 7 are stored, and it is also retrieved prior to nodes 5 and 7. It is a typical "First In, First Out" order, so the container is essentially a queue. After node 10 is printed, its two children, nodes 9 and 11, are stored in the container, too.

Nodes 5, 7, 9, and 11 do not have children, and they are printed from the head of the queue one by one. The printing process is summarized in Table 6-4.

Table 6-4. *The Process to Print the Binary Tree in Figure 6-10 from Top to Bottom*

Step	Operation	Nodes in the Queue
1	Print Node 8	6, 10
2	Print Node 6	10, 5, 7
3	Print Node 10	5, 7, 9, 11
4	Print Node 5	7, 9, 11
5	Print Node 7	9, 11
6	Print Node 9	11
7	Print Node 11	

Let's summarize the rules needed to print a binary tree from top level to bottom level. Once a node is printed, if it has children, its children nodes are stored in a queue. We continue to print the head of the queue, pop it from the queue, and push its children to the tail of the queue until there are no nodes left in the queue. The sample code in Listing 6-12 is based on the `queue` class in the standard template library (STL) in C++.

Listing 6-12. *C++ Code to Print a Binary Tree Level by Level*

```
void PrintFromTopToBottom(BinaryTreeNode* pRoot) {
    if(pRoot == NULL)
        return;

    std::queue<BinaryTreeNode *> queueTreeNodes;
    queueTreeNodes.push(pRoot);
```

```
    while(queueTreeNodes.size() > 0) {
        BinaryTreeNode *pNode = queueTreeNodes.front();
        queueTreeNodes.pop();

        printf("%d ", pNode->m_nValue);

        if(pNode->m_pLeft)
            queueTreeNodes.push(pNode->m_pLeft);

        if(pNode->m_pRight)
            queueTreeNodes.push(pNode->m_pRight);
    }
}
```

Source Code:

```
057_PrintTreeByLevel.cpp
```

Test Cases:

- Functional Cases (Normal binary trees with multiple levels)
- Boundary Cases (Special binary trees, such a binary tree with only one node, or all nodes in a binary tree only having left/right subtrees)
- Cases for Robustness (The pointer to the tree root is `NULL`)

■ **Question 58** How do you print a binary tree by level, in top down order, with each level in a line? Nodes in a level should be printed from left to right.

For example, the result of printing the binary tree in Figure 6-10 is:

8

6 10

5 7 9 11

This problem is quite similar to the preceding one for which a queue is utilized to store nodes to be printed. In order to print each level in a line, it is necessary to define two variables: one for the number of nodes to be printed in the current level, and the other for the number of nodes in the next level. The sample code is shown in Listing 6-13.

Listing 6-13. *C++ Code to Print a Binary Tree Level by Level*

```
void Print(BinaryTreeNode* pRoot) {
    if(pRoot == NULL)
        return;

    std::queue<BinaryTreeNode*> nodes;
    nodes.push(pRoot);
```

```
    int nextLevel = 0;
    int toBePrinted = 1;
    while(!nodes.empty()) {
        BinaryTreeNode* pNode = nodes.front();
        printf("%d ", pNode->m_nValue);

        if(pNode->m_pLeft != NULL) {
            nodes.push(pNode->m_pLeft);
            ++nextLevel;
        }
        if(pNode->m_pRight != NULL) {
            nodes.push(pNode->m_pRight);
            ++nextLevel;
        }

        nodes.pop();
        --toBePrinted;
        if(toBePrinted == 0) {
            printf("\n");
            toBePrinted = nextLevel;
            nextLevel = 0;
        }
    }
}
```

In the code above, the variable `toBePrinted` is the node count to be printed in the current level, and `nextLevel` is the node count on the next level. If a node has children nodes, its children are pushed into the queue, and `nextLevel` is increased by the number of children. When a node is printed, `toBePrinted` is decreased by one. When `toBePrinted` becomes zero, all nodes on the current level have been printed, and it moves on to print nodes on the next level.

Source Code:

`058_PrintTreeALevelInALine.cpp`

Test Cases:

- Functional Cases (Normal binary trees with multiple levels)
- Boundary Cases (Special binary trees, such as a binary tree with only one node, all nodes in a binary tree that only have left/right subtrees)
- Cases for Robustness (The pointer to the tree root is `NULL`)

■ **Question 59** How do you print a binary tree by level in zigzag order, each level in a line? That is to say, nodes on the first level are printed from left to right, nodes on the second level are printed from right to left, nodes on the third level are printed from left to right again, and so on.

For example, the result from printing the binary tree in Figure 6-11 is:

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

The process of printing a binary tree in zigzag order is a bit tricky. If a candidate cannot find a solution in a short time, it is a good practice to try some examples to analyze the process step-by-step. Let's take the sample tree with four levels in Figure 6-11 as an example.

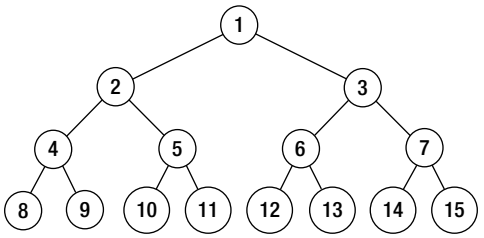

Figure 6-11. *A binary tree with four levels*

When the root node on the first level is printed, its left child (node 2) and right child (node 3) are stored into a data container. Node 3 is pushed into the data container after node 2, but node 3 should be printed before node 2 in zigzag order. Therefore, it sounds like the data container is a stack.

It continues to print nodes on the second level. Since node 3 is printed before node 2, children nodes of node 3 are stored into a data container prior to children nodes of node 2. According to the zigzag rule, children nodes of node 2 (node 4 and node 5) are printed before children nodes of node 3 (node 6 and node 7). Therefore, the data container used to store nodes in the third level is also a stack.

Table 6-5. *The First Seven Steps to Print the Binary Tree in Figure 6-11 in Zigzag Order. The Rightmost Numbers in Stack1 and Stack2 Are on the Top of Stacks.*

Step	Operation	Nodes in the Stack1	Nodes in the Stack2
1	Print Node 1	2, 3	
2	Print Node 3	2	7, 6
3	Print Node 2		7, 6, 5, 4
4	Print Node 4	8, 9	7, 6, 5
5	Print Node 5	8, 9, 10, 11	7, 7
6	Print Node 6	8, 9, 10, 11, 12, 13	7
7	Print Node 7	8, 9, 10, 11, 12, 13, 14, 15	

Additionally, since node 4 (the left child of node 2) should be printed before node 5 (the right child of node 2), the right children nodes in the third level should be pushed into the stack before their left sibling nodes. The stack for the third level is different from the stack for the second level, where the left child node (node 2) is pushed before its right sibling node (node 3).

After nodes in the second level are printed, it moves on to print nodes in the third node. The first node to be printed is node 4, and its children nodes (node 8 and node 9) at the fourth level are stored in a stack. Since node 9 should be printed prior to node 8, the left child is pushed before the right child again on the fourth level. The pushing order is different from the preceding level.

Therefore, two different stacks are needed to print a binary tree in zigzag order. The first stack is for nodes in the first and third levels, where left children nodes are pushed before right children nodes. The second stack is for nodes in the second and fourth level, where right children nodes are pushed before left children nodes.

Table 6-5 summarizes the first seven steps to print nodes. The next step to print nodes on the fourth level are quite simple because they are leaf nodes and do not have children. Nodes remaining in the first stack are popped and printed one by one.

It is time to develop code now that we have clear ideas about the process to print nodes in zigzag order. The sample code is shown in Listing 6-14.

Listing 6-14. *C++ Code to Print a Binary Tree in Zigzag Order*

```
void Print(BinaryTreeNode* pRoot) {
    if(pRoot == NULL)
        return;

    std::stack<BinaryTreeNode*> levels[2];
    int current = 0;
    int next = 1;

    levels[current].push(pRoot);
    while(!levels[0].empty() || !levels[1].empty()) {
        BinaryTreeNode* pNode = levels[current].top();
        levels[current].pop();

        printf("%d ", pNode->m_nValue);

        if(current == 0) {
            if(pNode->m_pLeft != NULL)
                levels[next].push(pNode->m_pLeft);
            if(pNode->m_pRight != NULL)
                levels[next].push(pNode->m_pRight);
        }
        else {
            if(pNode->m_pRight != NULL)
                levels[next].push(pNode->m_pRight);
            if(pNode->m_pLeft != NULL)
                levels[next].push(pNode->m_pLeft);
        }

        if(levels[current].empty()) {
            printf("\n");
```

```
            current = 1 - current;
            next = 1 - next;
        }
    }
}
```

Two stacks, `levels[0]` and `levels[1]`, are used in the code above. When it prints nodes in one stack, it stores children nodes in the other stack. After all nodes in a level are printed, it interchanges the two stacks and moves on to print the next level.

Source Code:

`059_PrintTreeZigzag.cpp`

Test Cases:

- Functional Cases (Normal binary trees with multiple levels)
- Boundary Cases (Special binary trees, such as a binary tree with only one node, all nodes in a binary tree only have left/right subtrees)
- Cases for Robustness (The pointer to the tree root is `NULL`)

Paths in Binary Trees

■ **Question 60** Given a binary tree and an integer value, please print all paths where the sum of node values equals the given integer. All nodes from the root node to a leaf node compose a path.

For example, given the binary tree in Figure 6-12, there are two paths with sum 22, of which one contains two nodes with values 10 and 12, and the other contains three nodes with values 10, 5, and 7.

Since paths always start from a root node, it might be easy to solve this problem with a traversal algorithm beginning from the root node in a binary tree. There are three common traversal orders, which are pre-order, in-order, and post-order. The solution first visits the root node with the pre-order traversal algorithm.

It visits node 5 after visiting the root node (10) of the binary tree in Figure 6-12 with the pre-order traversal. Since a binary tree node usually does not have a pointer to its parent, it is unknown what nodes have been visited when it reaches node 5 if the visited nodes on a path are not saved. Therefore, the current node is inserted into a path when it is reached during the traversal. The path contains two nodes with values 10 and 5 when it is visiting node 5. Then node 4 is inserted into the path, too, when it is reached. It arrives at a leaf node, and the sum of the three nodes in the path is 19. Since it is not the same as the target sum 22, the current path is not a qualified one.

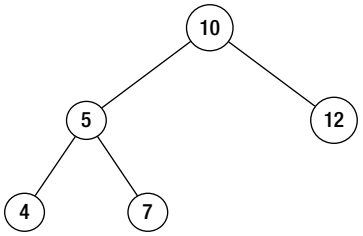

Figure 6-12. *A binary tree with two paths in which the sum of nodes is 22. One is the path through nodes 10, 5, and 7, and the other is the path through nodes 10 and 12.*

It continues to traverse other nodes. It has to return back to node 5, and then node 7 is visited. Notice that node 4 is no longer in the path from node 10 to node 7, so it should be removed from the path. When node 7 is visited, it is inserted into the path, which contains three nodes now. Since the sum of the value of these three nodes is 22, a qualified path has been found.

Last, the solution is going to visit node 12. It has to return back to node 5 and then back to node 10 before it visits node 12. When it returns back from a child node to its parent node, the child node is removed from the path. When it arrives at node 12 eventually, the path contains two nodes, one is node 10 and the other is node 12. Since the sum of the values of these two nodes is 22, the path is qualified, too. Table 6-6 summarizes this whole process.

Let's summarize some rules from the step-by-step analysis. When a node is visited with the pre-order traversal algorithm, it is inserted into the path, and the sum is increased by its value. When the node is a leaf and the sum is the same as specified, the path is qualified and it is printed. It continues to visit children nodes if the current node is not a leaf. After it finishes traversing a path to a leaf node, a recursive function will return back to its parent node automatically, so it has to remove the current node from the path before the function returns in order to make sure the nodes in the path correspond to the path from root node to its parent node.

The data structure to save paths should be a stack because paths should be consistent to the recursion status, and recursion is essentially pushing and popping in a call stack.

Table 6-6. *The Process to Traverse the Binary Tree in Figure 6-12 and Save the Sum of Node Values on Paths*

Step	Operation	A Leaf?	Path	Sum on Path
1	Visit Node 10	No	Node 10	10
2	Visit Node 5	No	Node 10, Node 5	15
3	Visit Node 4	Yes	Node 10, Node 5, Node 4	19
4	Return to Node 5		Node 10, Node 5	15
5	Visit Node 7	Yes	Node 10, Node 5, Node 7	22
6	Return to Node 5		Node 10, Node 5	15
7	Return to Node 10		Node 10	10
8	Visit Node 12	Yes	Node 10, Node 12	22

Listing 6-15 contains some sample code for this problem.

Listing 6-15. *C++ Code to Get Paths with a Specified Sum*

```
void FindPath(BinaryTreeNode* pRoot, int expectedSum) {
    if(pRoot == NULL)
        return;

    std::vector<int> path;
    int currentSum = 0;
    FindPath(pRoot, expectedSum, path, currentSum);
}

void FindPath(BinaryTreeNode* pRoot, int expectedSum, std::vector<int>& path, int currentSum)
{
    currentSum += pRoot->m_nValue;
    path.push_back(pRoot->m_nValue);

    // Print the path is the current node is a leaf
    // and the sum of all nodes value is same as expectedSum
    bool isLeaf = pRoot->m_pLeft == NULL && pRoot->m_pRight == NULL;
    if(currentSum == expectedSum && isLeaf) {
        printf("A path is found: ");
        std::vector<int>::iterator iter = path.begin();
        for(; iter != path.end(); ++ iter)
            printf("%d\t", *iter);

        printf("\n");
    }

    // If it is not a leaf, continue visition its children
    if(pRoot->m_pLeft != NULL)
        FindPath(pRoot->m_pLeft, expectedSum, path, currentSum);
    if(pRoot->m_pRight != NULL)
        FindPath(pRoot->m_pRight, expectedSum, path, currentSum);

    // Before returning back to its parent, remove it from path,
    path.pop_back();
}
```

In this code, it saves the path into a `vector` in the standard template library (STL). It uses the function `push_back` to insert a node and `pop_back` to remove a node to assure that it follows the "First in, Last out" rule in the path. The reason it does not utilize a `stack` in STL is that it can only get an element at the top of a stack, but it needs to get all nodes when it prints a path. Therefore, it simulates a stack with a `vector` rather than to utilize a `stack` directly.

Source Code:

`060_PathInTree.cpp`

Test Cases:

- Functional Cases (There are one or more paths with the specified sum; there are no paths with the specified sum)
- Boundary Cases (Special binary trees, such as a binary tree with only one node, or all nodes in a binary tree only have left/right subtrees)
- Cases for Robustness (The pointer to the tree root is `NULL`)

Divide and Conquer

The divide-and-conquer approach is an important algorithm design paradigm to analyze and solve complicated problems. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more similar subproblems until the subproblems become simple enough to be solved directly. The solutions to the subproblems are then combined to compose a solution to the original problem.

Take the problem to convert a binary search tree to a sorted double-linked list as an example. The problem is to convert a binary search tree; it can be divided into two subproblems to convert the left and right subtrees. After two subtrees are converted to two sorted double-linked lists, they are connected to the root node and the original tree is converted. More details are available in the section *Binary Search Trees and Double-linked Lists*.

The divide-and-conquer algorithm also works when it reduces a problem to only one subproblem. For example, if the problem is to permute characters in a string, it can be divided into a subproblem to permute all characters excluding the first one. Please refer to the section *Permutation and Combination* for more details about this problem.

Traversal Sequences and Binary Trees

■ **Question 61** Please build a binary tree with a pre-order traversal sequence and an in-order traversal sequence. All elements in these two given sequences are unique.

For example, if the input pre-order traversal sequence is {1, 2, 4, 7, 3, 5, 6, 8} and in-order traversal order is {4, 7, 2, 1, 5, 3, 8, 6}, the built tree is shown in Figure 6-13.

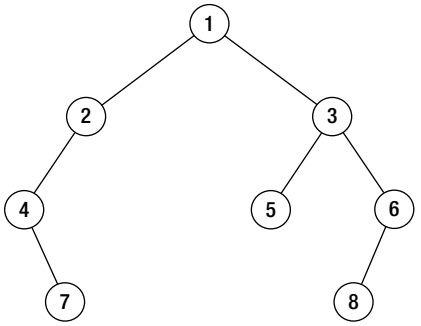

Figure 6-13. *The binary tree built from pre-order traversal sequence {1, 2, 4, 7, 3, 5, 6, 8} and in-order traversal sequence {4, 7, 2, 1, 5, 3, 8, 6}*

The root value is the first number in the pre-order traversal sequence of a binary tree, but it is in the middle of the in-order traversal sequence. Therefore, we have to scan the in-order traversal sequence to get the root value. As shown in Figure 6-14, the first number in the pre-order traversal sequence (number 1) is the root value. The location of root value is found if the whole in-order traversal sequence is scanned.

In the in-order traversal sequence, all numbers at the left side of the root value are for nodes in the left subtree, and all numbers at the right side of the root value correspond to nodes in the right subtree. According to the in-order traversal sequence in Figure 6-14, we find that there are three nodes in the left subtree and four in the right subtree. Therefore, the next three numbers behind the root value in the pre-order traversal sequence are for the left subtree, and the following four are for the right subtree.

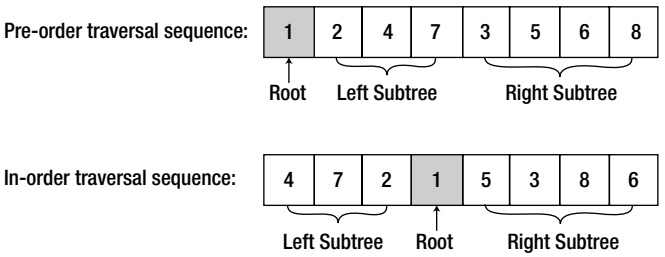

Figure 6-14. *Find the root node, the nodes in the left subtree, and the nodes in the right subtree in the pre-order traversal sequence and in-order traversal sequence.*

After the pre-order and in-order traversal sequences for both left and right subtrees are identified, subtrees can be built with recursion, similar to the process of building the whole tree. The sample code is shown in Listing 6-16 based on the divide-and-conquer approach.

Listing 6-16. *C++ Code to Build Binary Trees from Traversal Sequences*

```
BinaryTreeNode* Construct(int* preorder, int* inorder, int length) {
    if(preorder == NULL || inorder == NULL || length <= 0)
        return NULL;
```

```
    return ConstructCore(preorder, preorder + length - 1,
        inorder, inorder + length - 1);
}

BinaryTreeNode* ConstructCore(int* startPreorder, int* endPreorder, int* startInorder, int*
endInorder) {

    // The first number in the pre-order traversal sequence is the root value
    int rootValue = startPreorder[0];
    BinaryTreeNode* root = new BinaryTreeNode();
    root->m_nValue = rootValue;
    root->m_pLeft = root->m_pRight = NULL;

    if(startPreorder == endPreorder) {
        if(startInorder == endInorder
            && *startPreorder == *startInorder)
            return root;
        else
            throw std::exception("Invalid input.");
    }

    // Get the root value in the in-order traversal sequence
    int* rootInorder = startInorder;
    while(rootInorder <= endInorder && *rootInorder != rootValue)
        ++ rootInorder;

    if(rootInorder == endInorder && *rootInorder != rootValue)
        throw std::exception("Invalid input.");

    int leftLength = rootInorder - startInorder;
    int* leftPreorderEnd = startPreorder + leftLength;
    if(leftLength > 0) {
        // Build left subtree
        root->m_pLeft = ConstructCore(startPreorder + 1,
            leftPreorderEnd, startInorder, rootInorder - 1);
    }
    if(leftLength < endPreorder - startPreorder) {
        // Build rigth subtree
        root->m_pRight = ConstructCore(leftPreorderEnd + 1,
            endPreorder, rootInorder + 1, endInorder);
    }

    return root;
}
```

In the function `ConstructCore`, first the root value is gotten in the pre-order traversal sequence, and then it is located in the in-order traversal sequence. After the subsequence for the left and right subtrees are partitioned in the pre-order and in-order traversal sequences, the function `ConstructCore` is called recursively to build the left and right subtrees.

Source Code:

`061_ConstructBinaryTree.cpp`

Test Cases:

- Functional Cases (Sequences for normal binary trees, including some full binary trees; no binary trees can be built from the given sequences)
- Boundary Cases (Special binary trees, such as a binary tree with only one node, or all nodes in a binary tree only have left/right subtrees)
- Cases for Robustness (The pointers to one or two arrays are `NULL`)

■ **Question 62** How do you serialize and deserialize binary trees?

As discussed before, a binary tree can be built from a pre-order traversal sequence and an in-order traversal sequence. Inspired by this conclusion, if a binary is serialized to two number arrays for the pre-order and in-order traversal sequences, it can be rebuilt from these two arrays during deserialization.

This solution works, but has two disadvantages. One disadvantage is that there should not be any duplicated numbers in the array; otherwise, it causes problems during deserialization. The other disadvantage is that it has to read all numbers in the two traversal sequences before deserialization begins, so it has to wait for a long time to start deserialization if data comes from a stream. Let's explore other alternatives.

If serialization of a binary tree starts from the root node, the corresponding deserialization can start once the root value is received. Therefore, the pre-order traversal algorithm is applied to serialize a binary tree, which visits the root node first. When a `NULL` pointer is reached during traversal, it is serialized as a special character (such as a '$'). Additionally, node values should be separated by another character (such as a comma ','). According to these serialization rules, the binary tree in Figure 6-15 is serialized as a sequence "1,2,4,$,$,$,3,5,$,$,6,$,$". (See Listing 6-17.)

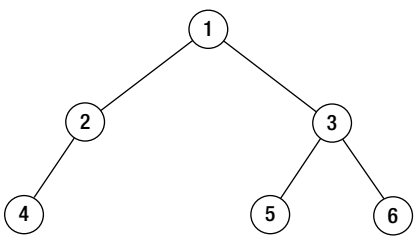

Figure 6-15. *A binary tree whose serialization sequence is "1,2,4,$,$,$,3,5,$,$,6,$,$"*

Listing 6-17. *C++ Code to Serialize Binary Trees*

```
void Serialize(BinaryTreeNode* pRoot, ostream& stream) {
    if(pRoot == NULL) {
        stream << "$,";
        return;
    }

    stream << pRoot->m_nValue << ',';
```

```
    Serialize(pRoot->m_pLeft, stream);
    Serialize(pRoot->m_pRight, stream);
}
```

Let's move on to analyze how to deserialize a binary tree. Take the serialization sequence "1,2,4,$,$,$,3,5,$,$,6,$,$" as an example. It first reads a number 1 from the stream, which is the root value of a binary tree. Another number 2 is read. Since the binary tree is serialized with the pre-order traversal algorithm, the number 2 is taken as the left child of the root node. Similarly, the next number, 4, is deserialized as the left child of the node with value 2. The following data read from the stream are two '$'s, which means that the node with value 4 is a leaf and does not have children nodes. It returns back to the parent node, the node with value 2, and continues to deserialize its right child. The right child of the node with value 2 is `NULL` because the next character read from the stream is also a '$'.

It returns back to the root node to deserialize the right subtree. A number 3 is read, so the right child of the root node contains value 3. The following data from the stream is another number 5, which is taken as the left child value of the node with value 3. The following two '$'s indicate that the node with value 5 is a leaf node. It continues to deserialize the right child of the node with value 3, which is another leaf node with value 6 because the next data from the stream is "6,$,$". It stops deserialization because all children of all non-leaf nodes have been constructed.

Listing 6-18 is the sample code used to deserialize a binary tree in C++.

Listing 6-18. *C++ Code to Deserialize Binary Trees*

```
void Deserialize(BinaryTreeNode** pRoot, istream& stream) {
    int number;
    if(ReadStream(stream, &number)) {
        *pRoot = new BinaryTreeNode();
        (*pRoot)->m_nValue = number;
        (*pRoot)->m_pLeft = NULL;
        (*pRoot)->m_pRight = NULL;

        Deserialize(&((*pRoot)->m_pLeft), stream);
        Deserialize(&((*pRoot)->m_pRight), stream);
    }
}
```

The function `ReadStream` in this code reads a token (a number or a `$') from a stream at every step. When a token received from the stream is a number, it returns `true`.

Source Code:

`062_SerializeBinaryTree.cpp`

Test Cases:

- Functional Cases (Normal binary trees, including some full binary trees)
- Boundary Cases (Special binary trees, such as a binary tree with only one node, or all nodes in a binary tree only have left/right subtrees)
- Cases for Robustness (The pointer to the root node of a binary tree is `NULL`)

■ **Question 63** Please check whether it is possible for an array to be the post-order traversal sequence of a binary search tree. All numbers in the input array are unique.

For example, the array {5, 7, 6, 9, 11, 10, 8} is the post-order traversal sequence of the binary search tree in Figure 6-16. However, there is not a binary search tree whose post-order traversal sequence is {7, 4, 6, 5}.

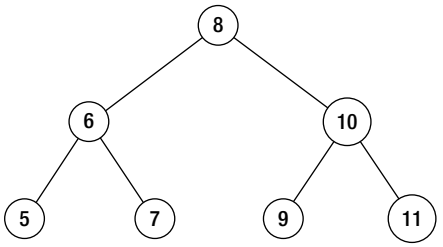

Figure 6-16. *A binary search tree built from a post-order traversal sequence {5, 7, 6, 9, 11, 10, 8}*

The last number in a post-order traversal sequence is the value of the root node. Other numbers in the sequence can be partitioned into two parts. The first numbers at the left side in the array, which are less than the value of the root node, are values of nodes in the left subtree; the following numbers, which are greater than the value of the root node, are values of nodes in the right subtree.

Take the input {5, 7, 6, 9, 11, 10, 8} as an example. The last number 8 in this sequence is the root value. The first three numbers (5, 7, and 6), which are less than 8, are values of nodes in the left subtree. The following three numbers (9, 11, and 10), which are greater than 8, are values of nodes in the right subtree.

The solution continues to construct the left subtree and right subtree according to the two sub-arrays with the same strategy. In the subsequence {5, 7, 6}, the last number 6 is the root value of the left subtree. The number 5 is the value of the left child since it is less than 6, and 7 is the value of the right child since it is greater than 6. Meanwhile, the last number 10 in the subsequence {9, 11, 10} is the root value of the right subtree. The number 9 is the value of the left child, and 11 is the value of right child accordingly.

Let's analyze another array {7, 4, 6, 5}. The last number 5 is the value of the root node. Since the first number 7 is greater than 5, there are no nodes in the left subtree, and the numbers 7, 4, 6 should be values of nodes in the right subtree. However, notice that the number 4, a value in the right subtree, is less than the root value 5. It violates the definition of binary search trees. Therefore, there are no binary search trees with the post-order traversal sequence {7, 4, 6, 5}.

It is not difficult to write code after we have examined this strategy. Some sample code is shown in Listing 6-19.

Listing 6-19. *C++ Code to Verify a Post-Order Traversal Sequence*

```
bool VerifySquenceOfBST(int sequence[], int length) {
    if(sequence == NULL || length <= 0)
        return false;

    int root = sequence[length - 1];
```

```
    // nodes in left sub-tree are less than root node
    int i = 0;
    for(; i < length - 1; ++ i) {
        if(sequence[i] > root)
            break;
    }

    // nodes in right sub-tree are greater than root node
    int j = i;
    for(; j < length - 1; ++ j) {
        if(sequence[j] < root)
            return false;
    }

    // Is left sub-tree a binary search tree?
    bool left = true;
    if(i > 0)
        left = VerifySquenceOfBST(sequence, i);

    // Is right sub-tree a binary search tree?
    bool right = true;
    if(i < length - 1)
        right = VerifySquenceOfBST(sequence + i, length - i - 1);

    return (left && right);
}
```

Source Code:

```
063_SequenceOfBST.cpp
```

Test Cases:

- Functional Cases (Post-order traversal sequences correspond to binary search trees, including full binary search trees; post-order traversal sequences do not correspond to any binary search trees)
- Boundary Cases (Special post-order traversal sequences where elements are increasingly or decreasingly sorted; there is only one element in the sequence)
- Cases for Robustness (The pointer to the array of traversal sequence is `NULL`)

Binary Search Trees and Double-Linked Lists

■ **Question 64** How do you convert a binary search tree into a sorted double-linked list without creating any new nodes? It is only allowed that you can reconnect links between existing nodes.

For example, after reconnecting links in the binary search tree in Figure 6-17(a), it gets a sorted double-linked list in Figure 6-17(b).

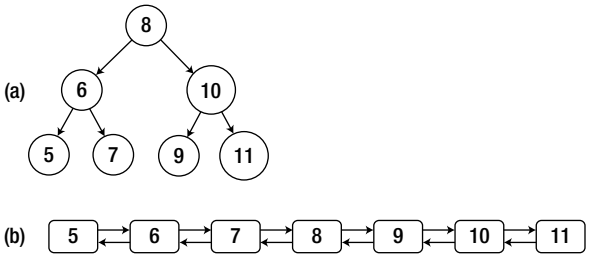

Figure 6-17. *A binary search tree and its converted sorted double-linked list*

Every node in a binary search tree has two links to its children nodes (some links might be `NULL` pointers). Every node in a double-linked list also has two links, one to the preceding node and the other to the next node. Additionally, the binary search tree is also a data structure for sorting, so a binary search tree can be converted to a sorted double-linked list in theory. The link to the left child in a binary search tree is connected to the preceding node in the converted double-linked list, and the link to the right child is connected to the next node.

Based on Divide and Conquer

If a binary search tree is scanned with the in-order traversal algorithm, nodes are traversed in increasing order of values. When the root node of the sample tree is visited, the tree can be viewed in three parts: the root value with value 8, the left subtree rooted at the node with value 6, and the right subtree rooted at the node with value 10. According to the sorting requirement, the root node with value 8 should be linked to the node in the left subtree with the maximal value (the node with value 7), as well as the node in the right subtree with the minimum value (the node with value 9), as shown in Figure 6-18.

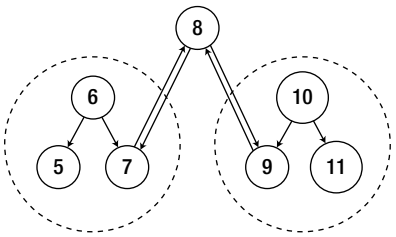

Figure 6-18. *Divide a binary tree into three parts: root node, left subtree, and right subtree. After the left subtree and right subtree are converted to double-linked lists, they are connected to the root node and the whole binary search tree is converted to a sorted double-linked list.*

Since the tree is scanned with the in-order traversal algorithm, the left subtree is converted when the root node is visited, and the last node in the converted double-linked list has the maximum value so far (node 7). The root node is connected to the list then, and it becomes the new last node in the list. We continue to convert the right subtree and connect the root node to the node with the minimum value in the right subtree (node 9). How do you convert the left and right subtrees? The process should be similar to the process to convert the whole tree, so it can be solved with recursion, as shown in Listing 6-20.

Listing 6-20. *C++ Code to Convert a BSTree to a Double-Linked List (Version 1)*

```
BinaryTreeNode* Convert(BinaryTreeNode* pRootOfTree) {
    BinaryTreeNode *pLastNodeInList = NULL;
    ConvertNode(pRootOfTree, &pLastNodeInList);

    // pLastNodeInList points to the last node in the doule-linked list,
    // but we are going to return the head node.
    BinaryTreeNode *pHeadOfList = pLastNodeInList;
    while(pHeadOfList != NULL && pHeadOfList->m_pLeft != NULL)
        pHeadOfList = pHeadOfList->m_pLeft;

    return pHeadOfList;
}

void ConvertNode(BinaryTreeNode* pNode, BinaryTreeNode** pLastNodeInList) {
    if(pNode == NULL)
        return;

    BinaryTreeNode *pCurrent = pNode;

    if (pCurrent->m_pLeft != NULL)
        ConvertNode(pCurrent->m_pLeft, pLastNodeInList);

    pCurrent->m_pLeft = *pLastNodeInList;
    if(*pLastNodeInList != NULL)
        (*pLastNodeInList)->m_pRight = pCurrent;

    *pLastNodeInList = pCurrent;

    if (pCurrent->m_pRight != NULL)
        ConvertNode(pCurrent->m_pRight, pLastNodeInList);
}
```

In this code, `pLastNodeInList` always points to the last node with the maximal value in the converted double-linked list. When the root node with value 8 is visited, the left subtree has been converted and `pLastNodeInList` points to the node with 7. The root node is linked to the list, and then `pLastNodeInList` points the root node with value 8. It takes `pLastNodeInList` as a parameter to continue converting the right subtree, and `pLastNodeInList` is connected to the node with value 9.

Based on Node Rotations

Node rotations are fundamental operations to maintain self-balancing binary trees, such as red-black trees. Let's borrow the ideas of node rotations to convert a binary search tree to a sorted double-linked list.

The conversion process begins from the root node. If the root node of a binary search tree has a left child, a right rotation occurs. As shown in Figure 6-19(b), the tree is rotated clockwise at the root node, 8. After the right rotation, node 6 (the left child of the original root node) becomes the new root, and node 7 (the right child of node 6) becomes the left child of node 8. Other links in the tree remain unchanged.

The tree is rotated clockwise at the new root node because it still has a left child. The left child node becomes the new root with a right rotation (Figure 6-19(c)). Since the new root, node 5, does not have a left child, no more rotations at the root are needed now. It traverses the tree along links to right children until the first node with a left child is found, that is, the node with value 8.

It rotates clockwise at node 8 (Figure 6-19(d)). Node 7, the left child of node 8, becomes the parent of node 8, and it is also linked to a right child of node 6, which is the original parent of node 8.

It continues to traverse on the tree along links to right children, and another node with left child (node 10) is found. The operation to rotate at node 10 is similar to the preceding steps, as shown in (Figure 6-19 (e)).

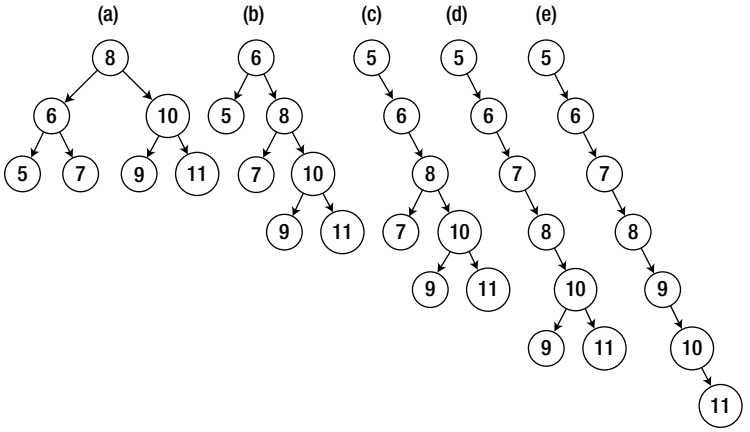

Figure 6-19. *The rotation process to convert a binary search tree to a sorted list. (a) The original binary search tree. (b) The tree with a right rotation at node 8. (c) The tree with a right rotation at node 6. (d) The tree with a right rotation at node 8. (e) The tree becomes a list with a right rotation at node 10.*

As shown in Figure 6-19(e), all nodes in the tree are linked as a single-linked list. Each child node is linked as a right child of its parent and links to left children are `NULL` pointers. In order to get a double-linked list, the last step is to link the parent node in the tree as the preceding node in the list.

The whole process to convert a binary search tree to a double-linked list can be implemented with C++, as shown in Listing 6-21.

Listing 6-21. *C++ Code to Convert a BSTree to a Double-Linked List (Version 2)*

```
BinaryTreeNode* Convert_solution2(BinaryTreeNode* pRoot) {
    BinaryTreeNode* pNode = pRoot;
```

```
    BinaryTreeNode* pHead = NULL;
    BinaryTreeNode* pParent = NULL;

    while(pNode != NULL) {
        while(pNode->m_pLeft != NULL) {
            // right rotation
            BinaryTreeNode* pLeft = pNode->m_pLeft;
            pNode->m_pLeft = pLeft->m_pRight;
            pLeft->m_pRight = pNode;
            pNode = pLeft;

            if(pParent != NULL)
                pParent->m_pRight = pNode;
        }

        if(pHead == NULL)
            pHead = pNode;

        pParent = pNode;
        pNode = pNode->m_pRight;
    }

    // build double-linked list
    BinaryTreeNode* pNode1 = pHead;
    if(pNode1 != NULL {
        BinaryTreeNode* pNode2 = pNode1->m_pRight;
        while(pNode2 != NULL) {
            pNode2->m_pLeft = pNode1;

            pNode1 = pNode2;
            pNode2 = pNode2->m_pRight;
        }
    }

    return pHead;
}
```

Source Code:

```
064_ConvertBinarySearchTree.cpp
```

Test Cases:

- Functional Cases (Normal binary search trees, including full binary search trees)
- Boundary Cases (Special binary search trees, such as a tree with only one node, or all nodes in a binary search tree only have left/right subtrees)
- Cases for Robustness (The pointer to the root node of a binary search tree is `NULL`)

Permutation and Combination

Question 65 Please print all permutations of a given string. For example, print “abc”, “acb”, “bac”, “bca”, “cab”, and “cba” when given the input string “abc”.

For many candidates, it is not a simple problem to get all permutations of a set of characters. In order to solve such a problem, candidates might try to divide it into simple subproblems. An input string is partitioned into two parts. The first part only contains the first characters, and the second part contains others. As shown in Figure 6-20, a string is divided into two parts with different background colors.

This solution gets permutations of a given string with two steps. The first step is to swap the first character with the following characters one by one. The second step is to get permutations of the string excluding the first character. Take the sample string “abc” as an example. It gets permutations of “bc” when the first character is ‘a’. It then swaps the first character with ‘b’, gets permutations of “ac”, and finally gets permutation of “ba” after swapping the first character with ‘c’.

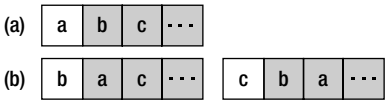

Figure 6-20. *The process to get permutations of a string. (a) A string is divided into two parts of which the first part only contains the first character, and the second part contains others (with gray background). (b) All characters in the second part are swapped with the first character one by one.*

The process to get permutations of a string excluding the first character is similar to the process to get permutations of a whole string. Therefore, it can be solved recursively, as shown in Listing 6-22.

Listing 6-22. *C Code to Get Permutations of a String*

```
void Permutation(char* pStr) {
    if(pStr == NULL)
        return;

    PermutationCore(pStr, pStr);
}

void PermutationCore(char* pStr, char* pBegin) {
    char *pCh = NULL;
    char temp;

    if(*pBegin == '\0') {
        printf("%s\n", pStr);
    }
    else {
        for(pCh = pBegin; *pCh != '\0'; ++ pCh) {
            temp = *pCh;
```

```
            *pCh = *pBegin;
            *pBegin = temp;

            PermutationCore(pStr, pBegin + 1);

            temp = *pCh;
            *pCh = *pBegin;
            *pBegin = temp;
        }
    }
}
```

Source Code:

`065_StringPermutation.c`

Test Cases:

- Functional Cases (The input string contains one or more characters)
- Cases for Robustness (The string is empty; the pointer to the string is `NULL`)

■ **Question 66** How many distinct ways are available to place eight queens on a chessboard, where there are no two queens that can attack each other? That is to say, there are no two queens located at the same row, same column, or same diagonal.

For example, a solution in Figure 6-21 places eight queens on an 8×8 chessboard where any queen cannot attack another.

An array `columnIndex`[8] is defined, of which the i^{th} number stands for the column index of the queen at the i^{th} row. The eight numbers in the array are initialized as numbers 0, 1, 2, 3, 4, 5, 6, 7, and then we try to get all permutations of the array. Since the array is initialized with eight different numbers, any two queens are in different columns. It is only necessary to check whether there are two queens on the same diagonal.

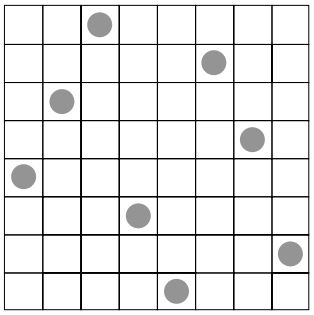

Figure 6-21. *Place eight queens (denoted as dots) on a chessboard where no two queens attack each other*

Therefore, the eight-queen puzzle can be solved with the code in Listing 6-23.

Listing 6-23. *C Code for the Eight Queens Puzzle*

```
int EightQueen() {
    int columnIndex[8] = {0, 1, 2, 3, 4, 5, 6, 7};
    int count = 0;
    Permutation(columnIndex, 8, 0, &count);

    return count;
}

void Permutation(int columnIndex[], int length, int index, int* count) {
    int i, temp;

    if(index == length) {
        if(Check(columnIndex, length) != 0) {
            (*count)++;
        }
    }
    else {
        for(i = index; i < length; ++ i) {
            temp = columnIndex[i];
            columnIndex[i] = columnIndex[index];
            columnIndex[index] = temp;

            Permutation(columnIndex, length, index + 1, count);

            temp = columnIndex[index];
            columnIndex[index] = columnIndex[i];
            columnIndex[i] = temp;
        }
    }
}

/* If there are two queens on the same diagonal, it returns 0,
   otherwise it returns 1. */
int Check(int columnIndex[], int length) {
    int i, j;

    for(i = 0; i < length; ++ i) {
        for(j = i + 1; j < length; ++ j) {
            if((i - j == columnIndex[i] - columnIndex[j])
                || (j - i == columnIndex[i] - columnIndex[j]))
            return 0;
        }
    }

    return 1;
}
```

Source Code:

066_EightQueens.c

■ **Question 67** There are *n* arrays. A permutation is generated when an element is selected from each array. How do you generate all permutations of *n* arrays?

For example, permutations for the 3 arrays {1, 2}, {3, 4}, {5, 6} are: {1, 3, 5}, {1, 3, 6}, {1, 4, 5}, {1, 4, 6}, {2, 3, 5}, {2, 3, 6}, {2, 4, 5}, and {2, 4, 6}.

Similar to finding permutations of a string, it divides the *n* input arrays into two parts: the first array and the remaining *n*-1 arrays. After an element is selected from the first array, it continues to find permutations of the remaining arrays. The process needed to find permutations of the next *n*-1 arrays is identical to the process to find permutations of all *n* arrays, so it can be solved recursively, as shown in Listing 6-24.

Listing 6-24. *Java Code to Permute Arrays*

```
void permute(ArrayList<int[]> arrays) {
    Stack<Integer> permutation = new Stack<Integer>();
    permuteCore(arrays, permutation);
}

void permuteCore(ArrayList<int[]> arrays, Stack<Integer> permutation) {
    if(permutation.size() == arrays.size()) {
        System.out.println(permutation);
        return;
    }

    int[] array = arrays.get(permutation.size());
    for(int i = 0; i < array.length; ++i) {
        permutation.push(array[i]);
        permuteCore(arrays, permutation);
        permutation.pop();
    }
}
```

Source Code:

067_ArrayPermutation.java

Test Cases:

- Functional Cases (Some normal arrays)
- Boundary Cases (There is only one array; there is only one number in each array)

■ **Question 68** Please generate all combinations of a given string. For example, combinations of a given string "abc" are "a", "b", "c", "ab", "ac", "bc", and "abc".

Based on Divide and Conquer

Suppose the length of a given string is n and we are going to find one of its combinations with m characters. All characters are divided into two parts: the first character and the other n-1 characters. There are two choices for the first characters. The first choice is to insert the first character in the combination, and it continues to select m-1 characters from the remaining n-1 characters in the string. The other choice is to ignore the first character and select m characters from the next n-1 characters. Therefore, combinations of a given string can be generated recursively, as shown in Listing 6-25.

Listing 6-25. *Java Code for String Combinations (Version 1)*

```
void combination(String str) {
    Stack<Character> result = new Stack<Character>();
    for(int i = 1; i <= str.length(); ++ i) {
        combination(str, 0, i, result);
    }
}

void combination(String str, int index, int number, Stack<Character> result) {
    if(number == 0) {
        System.out.println(result);
        return;
    }

    if(index == str.length())
        return;

    // select the character str[index]
    result.push(str.charAt(index));
    combination(str, index + 1, number - 1, result);
    result.pop();

    // ignore the character str[index]
    combination(str, index + 1, number, result);
}
```

Since a combination may contain one characters, two characters, …, or n characters for a given string with length n, there is a loop in the method `combination(String str)`.

Based on Bit Operations

Some characters are selected that compose a combination. A set of bits is utilized, where each bit stands for a character in the string. If the i^{th} character is selected for a combination, the i^{th} bit is `true`; otherwise,

it is `false`. For instance, three bits are used for combinations of the string "abc". If the first two characters 'a' and 'b' are selected to compose a combination "ab", the corresponding bits are {1, 1, 0}. Similarly, bits corresponding to another combination "ac" are {1, 0, 1}. We are able to get all combinations of a string with length *n* if we can get all possible combinations of *n* bits.

A number is composed of a set of bits. All possible combinations of *n* bits correspond to numbers from 1 to 2^n-1. Therefore, each number in the range between 1 and 2^n-1 corresponds to a combination of a string with length *n*. For example, the number 6 is composed of bits {1, 1, 0}, so the first and second characters are selected in the string "abc" to generate the combination "ab". Similarly, the number 5 with bits {1, 0, 1} corresponds to the combination "ac".

A new solution is found that generates combinations with *n* characters based on a set of bits corresponding to numbers from 1 to 2^n-1. The sample code is shown in Listing 6-26.

Listing 6-26. *Java Code for String Combinations (Version 2)*

```
void combination_solution2(String str) {
    BitSet bits = new BitSet(str.length());
    while(increment(bits, str.length()))
        print(str, bits);
}

boolean increment(BitSet bits, int length) {
    int index = length - 1;

    while(index >= 0 && bits.get(index)) {
        bits.clear(index);
        --index;
    }

    if(index < 0)
        return false;

    bits.set(index);
    return true;
}

void print(String str, BitSet bits) {
    for(int i = 0; i < str.length(); ++i) {
        if(bits.get(i))
            System.out.print(str.charAt(i));
    }

    System.out.println();
}
```

The method `increment` increases a number represented in a set of bits. The algorithm clears 1 bits from the rightmost bit until a 0 bit is found. It then sets the rightmost 0 bit to 1. For example, in order to increase the number 5 with bits {1, 0, 1}, it clears 1 bits from the right side and sets the rightmost 0 bit to 1. The bits become {1, 1, 0} for the number 6, which is the result of increasing 5 by 1.

Source Code:

`068_StringCombination.java`

Test Cases:

- Functional Cases (Some normal strings)
- Boundary Cases (An empty string)

Summary

It is impossible to avoid difficult problems during interviewers. Figures, examples, and the divide-and-conquer approach are three strategies to analyze and solve complicated coding problems.

Figures are helpful tools to visualize difficult problems, especially when the problems are related to data structures such as lists, binary trees, and 2D arrays.

Examples are quite useful to solve problems about abstract algorithms. It is much easier to uncover hidden rules if candidates analyze the step-by-step process with some sample inputs.

The divide-and-conquer approach is an effective strategy to solve complex problems. It divides them into subproblems, solves subproblems recursively, and combines solutions of subproblems into a solution of the original problem.