

**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.

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 ofStacks” (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. Ifoperation 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 minimumnumber 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 minimumnumber. When a new number that is less than the minimumnumber is pushed, the minimumgets updated. It sounds good. However, how do you get the next minimumwhen the current minimumis 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 minimumnumber. When the minimumis popped, the solution should be able to find the next minimum. Therefore, it is necessary to restore the next minimumnumber after the current minimumone 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 minimumnumber 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 ofthe auxiliary stack is always the minimumnumber 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 fromthe auxiliary stack. If the minimumnumber is popped, the next minimumnumber should also be on the top ofthe auxiliary stack. In the fifth step, it pops 1 fromthe data stack, and it also pops the number on the top ofthe auxiliary, which is 1. The next minimumnumber, 2, is now on the top ofthe auxiliary stack. Ifit continues popping fromboth 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 ofthe auxiliary stack. Ifa new number, 0, is pushed into the data stack, as well as the auxiliary stack, the number on top ofthe auxiliary stack is the minimum0.

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  $2 \times \text{value} - \text{min}$  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  $2 \times \text{min} - \text{top}$ .

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  $2 \times \text{value} - \text{min}$ . Notice that  $2 \times \text{value} - \text{min} = \text{value} + (\text{value} - \text{min})$ , and  $\text{value} + (\text{value} - \text{min})$  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  $\text{top} = 2 \times \text{value} - \text{min}'$  and *value* is the current *min*, the previous minimum number is restored with  $\text{min}' = 2 \times \text{min} - \text{top}$ .

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 to 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.

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.

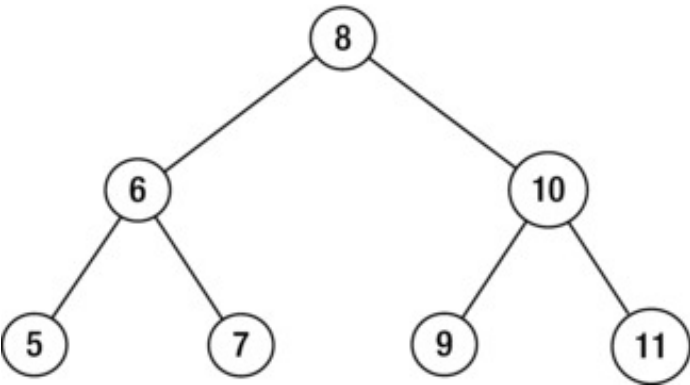


Figure 6-10. A sample binary tree with three levels

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*> queueTreeNode;  
    queueTreeNode.push(pRoot);  
  
    while(queueTreeNode.size() > 0) {  
  
        BinaryTreeNode *pNode = queueTreeNode.front();  
        queueTreeNode.pop();  
  
        printf("%d ", pNode->m_nValue);  
  
        if(pNode->m_pLeft)  
            queueTreeNode.push(pNode->m_pLeft);  
  
        if(pNode->m_pRight)  
            queueTreeNode.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:

05B\_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.

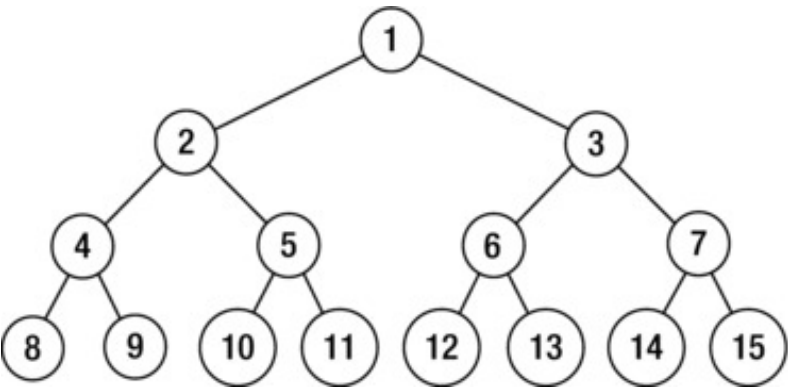


Figure 6-11. A binary tree with four levels

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.

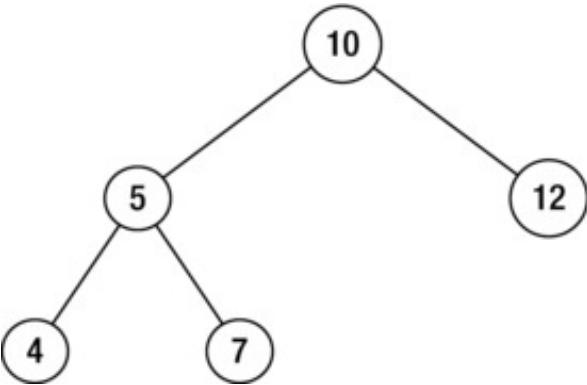


Figure 6-12. A binary tree with two paths in which the sum of nodes is 22. One is the path through nodes 10, 5, and 7, and the other is the path through nodes 10 and 12.

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 visitation 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` )