

n 位将被丢弃。但右移时处理最左边位的情形要稍微复杂一点。如果数字是一个无符号数值，则用0填补最左边的 n 位；如果数字是一个有符号数值，则用数字的符号位填补最左边的 n 位。也就是说，如果数字原先是一个正数，则右移之后在最左边补 n 个0；如果数字原来是负数，则右移之后在最左边补 n 个1。下面是对两个8位有符号数进行右移的例子：

`00001010 >> 2 = 00000010`

`10001010 >> 3 = 11110001`

面试题15“二进制中1的个数”就是直接考查位运算的例子，而面试题56“数组中数字出现的次数”、面试题65“不用加减乘除做加法”等都是根据位运算的特点来解决问题的。

面试题15：二进制中1的个数

题目：请实现一个函数，输入一个整数，输出该数二进制表示中1的个数。例如，把9表示成二进制是1001，有2位是1。因此，如果输入9，则该函数输出2。

❖ 可能引起死循环的解法

这是一道很基本的考查二进制和位运算的面试题。题目不是很难，面试官提出问题之后，我们很快就能形成一个基本的思路：先判断整数二进制表示中最右边一位是不是1；接着把输入的整数右移一位，此时原来处于从右边数起的第二位被移到最右边了，再判断是不是1；这样每次移动一位，直到整个整数变成0为止。现在的问题变成了怎么判断一个整数的最右边是不是1。这很简单，只要把整数和1做位与运算看结果是不是0就知道了。1除最右边的一位之外所有位都是0。如果一个整数与1做与运算的结果是1，则表示该整数最右边一位是1，否则是0。基于这种思路，我们很快就能写出如下代码：

```
int NumberOf1(int n)
{
    int count = 0;
    while(n)
    {
        if(n & 1)
            count++;
        n >>= 1;
    }
}
```

```

        n = n >> 1;
    }

    return count;
}

```

面试官看了代码之后可能会问：把整数右移一位和把整数除以 2 在数学上是等价的，那上面的代码中可以把右移运算换成除以 2 吗？答案是否定的。因为除法的效率比移位运算要低得多，在实际编程中应尽可能地用移位运算符代替乘除法。

面试官接下来可能要问的第二个问题就是：上面的函数如果输入一个负数，比如 0x80000000，则运行的时候会发生什么情况？当把负数 0x80000000 右移一位的时候，并不是简单地把最高位的 1 移到第二位变成 0x40000000，而是 0xC0000000。这是因为移位前是一个负数，仍然要保证移位后是一个负数，因此移位后的最高位会设为 1。如果一直做右移运算，那么最终这个数字就会变成 0xFFFFFFFF 而陷入死循环。

◆ 常规解法

为了避免死循环，我们可以不右移输入的数字 n 。首先把 n 和 1 做与运算，判断 n 的最低位是不是为 1。接着把 1 左移一位得到 2，再和 n 做与运算，就能判断 n 的次低位是不是 1……这样反复左移，每次都能判断 n 的其中一位是不是 1。基于这种思路，我们可以把代码修改如下：

```

int NumberOf1(int n)
{
    int count = 0;
    unsigned int flag = 1;
    while(flag)
    {
        if(n & flag)
            count++;

        flag = flag << 1;
    }

    return count;
}

```

在这个解法中，循环的次数等于整数二进制的位数，32 位的整数需要循环 32 次。下面再介绍一种算法，整数中有几个 1 就只需要循环几次。

❖ 能给面试官带来惊喜的解法

在分析这种算法之前，我们先来分析把一个数减去 1 的情况。如果一个整数不等于 0，那么该整数的二进制表示中至少有一位是 1。先假设这个数的最右边一位是 1，那么减去 1 时，最后一位变成 0 而其他所有位都保持不变。也就是最后一位相当于做了取反操作，由 1 变成了 0。

接下来假设最后一位不是 1 而是 0 的情况。如果该整数的二进制表示中最右边的 1 位于第 m 位，那么减去 1 时，第 m 位由 1 变成 0，而第 m 位之后的所有 0 都变成 1，整数中第 m 位之前的所有位都保持不变。举个例子：一个二进制数 1100，它的第二位是从最右边数起的一个 1。减去 1 后，第二位变成 0，它后面的两位 0 变成 1，而前面的 1 保持不变，因此得到的结果是 1011。

在前面两种情况中，我们发现把一个整数减去 1，都是把最右边的 1 变成 0。如果它的右边还有 0，则所有的 0 都变成 1，而它左边的所有位都保持不变。接下来我们把一个整数和它减去 1 的结果做位与运算，相当于把它最右边的 1 变成 0。还是以前面的 1100 为例，它减去 1 的结果是 1011。我们再把 1100 和 1011 做位与运算，得到的结果是 1000。我们把 1100 最右边的 1 变成了 0，结果刚好就是 1000。

我们把上面的分析总结起来就是：把一个整数减去 1，再和原整数做与运算，会把该整数最右边的 1 变成 0。那么一个整数的二进制表示中有多少个 1，就可以进行多少次这样的操作。基于这种思路，我们可以写出新的代码：

```
int NumberOf1(int n)
{
    int count = 0;

    while (n)
    {
        ++count;
        n = (n - 1) & n;
    }

    return count;
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/15_NumberOf1InBinary



测试用例：

- 正数（包括边界值 1、0x7FFFFFFF）。
- 负数（包括边界值 0x80000000、0xFFFFFFFF）。
- 0。



本题考点：

- 考查应聘者对二进制及位运算的理解。
- 考查应聘者分析、调试代码的能力。如果应聘者在面试过程中采用的是第一种思路，则当面试官提示他输入负数将会出现问题时，面试官会期待他能在心中运行代码，自己找出运行出现死循环的原因。这就要求应聘者有一定的调试功底。



相关题目：

- 用一条语句判断一个整数是不是 2 的整数次方。一个整数如果是 2 的整数次方，那么它的二进制表示中有且只有一位是 1，而其他所有位都是 0。根据前面的分析，把这个整数减去 1 之后再和它自己做与运算，这个整数中唯一的 1 就会变成 0。
- 输入两个整数 m 和 n ，计算需要改变 m 的二进制表示中的多少位才能得到 n 。比如 10 的二进制表示为 1010，13 的二进制表示为 1101，需要改变 1010 中的 3 位才能得到 1101。我们可以分为两步解决这个问题：第一步求这两个数的异或；第二步统计异或结果中 1 的位数。



举一反三：

把一个整数减去 1 之后再和原来的整数做位与运算，得到的结果相当于把整数的二进制表示中最右边的 1 变成 0。很多二进制的问题都可以用这种思路解决。

2.5 本章小结

本章着重介绍了应聘者在面试之前应该认真准备的基础知识。为了应对编程面试，应聘者需要在编程语言、数据结构和算法3个方面做好准备。

面试官通常采用概念题、代码分析题及编程题这3种常见题型来考查应聘者对某一编程语言的掌握程度。本章的2.2节讨论了C++/C#语言这3种题型的常见面试题。

数据结构题目一直是面试官考查的重点。数组和字符串是两种最基本的数据结构。链表应该是面试题中使用频率最高的一种数据结构。如果面试官想加大面试的难度，那么他很有可能会选用与树（尤其是二叉树）相关的面试题。由于栈与递归调用密切相关，队列在图（包括树）的宽度优先遍历中需要用到，因此应聘者也需要掌握这两种数据结构。

算法是面试官喜欢考查的另外一个重点。查找（特别是二分查找）和排序（特别是快速排序和归并排序）是面试中经常考查的算法，应聘者一定要熟练掌握。回溯法很适合解决迷宫及其类似的问题。如果面试题是求一个问题的最优解，那么可以尝试使用动态规划。假如我们在用动态规划分析问题时发现每一步都存在一个能得到最优解的选择，那么可以尝试使用贪心算法。另外，应聘者还要掌握分析时间复杂度的方法，理解即使同一思路，基于循环和递归的不同实现，它们的时间复杂度可能大不相同。很多时候我们会用自上而下的递归思路分析问题，却会基于自下而上的循环实现代码。

位运算是针对二进制数字的运算规律。只要应聘者熟练掌握了二进制的与、或、异或运算及左移、右移操作，就能解决与位运算相关的面试题。

第3章

高质量的代码

3.1 面试官谈代码质量

“一般会考查应聘人员对代码的容错处理能力，对一些特别的输入会询问应聘人员是否考虑、如何处理。不能容忍代码只是针对一种假想的‘正常值’进行处理，不考虑异常状况，也不考虑资源的回收等问题。”

——殷焰（支付宝，高级安全测试工程师）

“如果是因为粗心犯错，则可以原谅，因为毕竟面试的时候会紧张；不能容忍的是，该掌握的知识点却没有掌握，而且提醒了还不知道。比如下面的：

```
double d1, d2;  
...  
if(d1==d2)  
..."
```

——马凌洲（Autodesk，软件开发经理）

¹ 由于精度原因，不能用等号判断两个小数是否相等，请参考面试题 26 “树的子结构”。

“最不能容忍功能错误，忽略边界情况。”

——尹彦（英特尔，软件工程师）

“如果一个程序员连变量、函数命名都毫无章法，解决一个具体问题都找不到一个最合适的数据结构，那么这会让面试官对他的印象大打折扣，因为这只能说明他程序写得太少，不够熟悉。”

——吴斌（英伟达，图形设计师）

“我会从程序的正确性和鲁棒性两方面检验代码的质量。会关注对输入参数的检查、处理错误和异常的方式、命名方式等。对于没有工作经验的学生，程序正确性之外的错误基本都能容忍，但经过提示后希望能够很快解决。对于有工作经验的人，不能容忍考虑不周到、有明显的鲁棒性错误。”

——田超（微软，SDE II）

3.2 代码的规范性

面试官是根据应聘者写出的代码来决定是否录用他的。如果应聘者代码写得不够规范，影响面试官阅读代码的兴致，那么面试官就会默默地减去几分。如图3.1所示，书写、布局和命名都决定着代码的规范性。

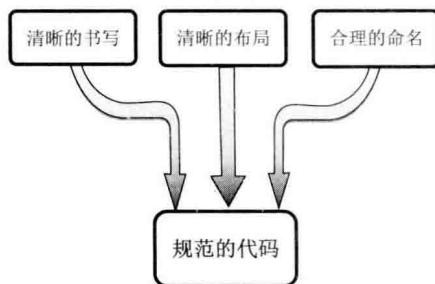


图3.1 影响代码规范性的因素：书写、布局和命名

首先，规范的代码书写清晰。绝大部分面试都是要求应聘者在白纸或

者白板上书写。由于现代人已经习惯了敲键盘打字，手写变得越来越不习惯，因此写出来的字潦草难辨。虽然应聘者没有必要为了面试特意去练字，但在面试过程中减慢写字的速度，尽量把每个字母写清楚还是很有必要的。不用担心没有时间去写代码，通常编程面试的代码量都不会超过 50 行，书写不用花多少时间，关键是在写代码之前形成清晰的思路并能把思路用编程语言清楚地书写出来。

其次，规范的代码布局清晰。平时程序员在集成开发环境如 Visual Studio 里面写代码，依靠专业工具调整代码的布局，加入合理的缩进并让括号对齐成对呈现。离开了这些工具手写代码，我们就要格外注意布局问题。当循环、判断较多，逻辑较复杂时，缩进的层次可能会比较多。如果布局不够清晰，缩进也不能体现代码的逻辑，那么面试官面对这样的代码将会头昏脑涨。

最后，规范的代码命名合理。很多初学编程的人在写代码时总是习惯用最简单的名字来命名，变量名是 i、j、k，函数名是 f、g、h。由于这样的名字不能告诉读者对应的变量或者函数的意义，代码一长就会变得晦涩难懂。强烈建议应聘者在写代码的时候，用完整的英文单词组合命名变量和函数，比如函数需要传入一棵二叉树的根节点作为参数，则可以把该参数命名为 `BinaryTreeNode* pRoot`，不要因为这样会多写几个字母而觉得麻烦。如果一眼能看出变量、函数的用途，应聘者就能避免自己搞混淆而犯一些低级的错误。同时合理的命名也能让面试官一眼就能读懂代码的意图，而不是让他去猜变量 m 到底是数组中的最大值还是最小值。



面试小提示：

应聘者在写代码的时候，最好用完整的英文单词组合命名变量和函数，以便面试官能一眼读懂代码的意图。

3.3 代码的完整性

在面试过程中，面试官会非常关注应聘者考虑问题是否周全。面试官通过检查代码是否完整来考查应聘者的思维是否全面。通常面试官会检查应聘者的代码是否完成了基本功能、输入边界值是否能得到正确的输出、是否对各种不合规范的非法输入做出了合理的错误处理。

1. 从3个方面确保代码完整性

应聘者在写代码之前，首先要把可能的输入都想清楚，从而避免在程序中出现各种各样的质量漏洞。也就是说，在编码之前要考虑单元测试。如果能够设计全面的单元测试用例并在代码中体现出来，那么写出的代码自然也就是完整正确的了。通常我们可以从功能测试、边界测试和负面测试3个方面设计测试用例，以确保代码的完整性，如图3.2所示。

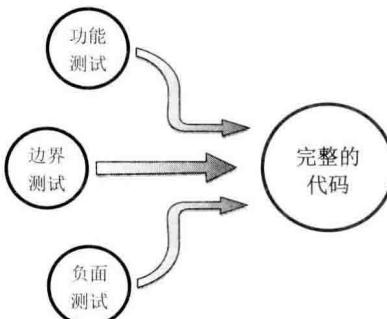


图3.2 从功能测试、边界测试和负面测试3个方面设计测试用例，以确保代码的完整性

首先要考虑的是普通功能测试的测试用例。我们首先要保证写出的代码能够完成面试官要求的基本功能。比如面试题要求完成的功能是把字符串转换成整数，我们就可以考虑输入字符串"123"来测试自己写的代码。这里要把零、正数和负数都考虑进去。

在考虑功能测试的时候，我们要尽量突破常规思维的限制。面试的时候我们经常受到惯性思维的限制，从而看不到更多的功能需求。比如面试题17“打印从1到最大的n位数”，很多人觉得这道题很简单。最大的3位数是999、最大的4位数是9999，这些数字很容易就能算出来。但是最大的n位数都能用int型表示吗？超出int的范围我们可以考虑long long类型，超出long long能够表示的范围呢？面试官是不是要求考虑任意大的数字？如果面试官确认题目要求的是任意大的数字，那么这道题目就是一个大数问题，此时我们需要特殊的的数据结构来表示数字，比如用字符串或者数组来表示大的数字，以确保不会溢出。

其次需要考虑各种边界值的测试用例。很多时候我们的代码中都会有循环或者递归。如果我们的代码基于循环，那么结束循环的边界条件是否正确？如果基于递归，那么递归终止的边界值是否正确？这些都是边界测试时要考虑的用例。还是以字符串转换成整数的问题为例，我们写出的代

码应该确保能够正确转换最大的正整数和最小的负整数。

最后还需要考虑各种可能的错误输入，也就是通常所说的负面测试的测试用例。我们写出的函数除了要顺利地完成要求的功能，当输入不符合要求的时候还能做出合理的错误处理。在设计把字符串转换成整数的函数的时候，我们就要考虑当输入的字符串不是一个数字时，比如"1a2b3c"，该怎么告诉函数的调用者这个输入是非法的。

前面所说的都是要全面考虑当前需求对应的各种可能输入。在软件开发过程中，永远不变的就是需求会一直改变。如果我们在面试的时候写出的代码能够把将来需求可能的变化都考虑进去，在需求发生变化的时候能够尽量减少代码改动的风险，那么我们就向面试官展示了自己对程序可扩展性和可维护性的理解，通过面试就是水到渠成的事情了。请参考面试题 21 “调整数组顺序使奇数位于偶数前面” 中关于可扩展性和可维护性的讨论。

2. 3 种错误处理的方法

通常我们有 3 种方式把错误信息传递给函数的调用者。第一种方式是函数用返回值来告知调用者是否出错。比如很多 Windows 的 API 就是这个类型。在 Windows 中，很多 API 的返回值为 0 表示 API 调用成功，而返回值不为 0 表示在 API 的调用过程中出错了。微软为不同的非零返回值定义了不同的意义，调用者可以根据这些返回值判断出错的原因。这种方式最大的问题是使用不便，因为函数不能直接把计算结果通过返回值赋值给其他变量，同时也不能把这个函数计算的结果直接作为参数传递给其他函数。

第二种方式是当错误发生时设置一个全局变量。此时我们可以在返回值中传递计算结果了。这种方法比第一种方法使用起来更加方便，因为调用者可以直接把返回值赋值给其他变量或者作为参数传递给其他函数。Windows 的很多 API 运行出错之后，也会设置一个全局变量。我们可以通过调用函数 GetLastError 分析这个表示错误的全局变量，从而得知出错的原因。但这种方法有一个问题：调用者很容易忘记检查全局变量，因此在调用出错的时候忘记进行相应的错误处理，从而留下安全隐患。

第三种方式是异常。当函数运行出错的时候，我们就抛出一个异常，还可以根据不同的出错原因定义不同的异常类型。因此，函数的调用者根据异常的类型就能知道出错的原因，从而做出相应的处理。另外，我们能

显式划分程序正常运行的代码块（try模块）和处理异常的代码块（catch模块），逻辑比较清晰。异常在高级语言如C#中是强烈推荐的错误处理方式，但有些早期的语言如C语言还不支持异常。另外，在抛出异常的时候，程序的执行会打乱正常的顺序，对程序的性能有很大的影响。

上述3种错误处理的方式各有其优缺点，如表3.1所示。那么，面试的时候我们该采用哪种方式呢？这要看面试官的需求。在听到面试官的题目之后，我们要尽快分析出可能存在哪些非法的输入，并和面试官讨论该如何处理这些非法输入。

表3.1 返回值、全局变量和异常3种错误处理方式的优缺点比较

	优 点	缺 点
返回值	和系统API一致	不能方便地使用计算结果
全局变量	能够方便地使用计算结果	用户可能会忘记检查全局变量
异常	可以为不同的出错原因定义不同的异常类型，逻辑清晰明了	有些语言不支持异常，抛出异常时对性能有负面影响

面试题16：数值的整数次方

题目：实现函数double Power(double base, int exponent)，求base的exponent次方。不得使用库函数，同时不需要考虑大数问题。

我们都知道，在C语言的库中有一个pow函数可以用来求乘方，本题要求实现类似于pow的功能。要求实现特定库函数（特别是处理数值和字符串的函数）的功能是一类常见的面试题。在本书收集的面试题中，除这个题目外，还有面试题67“把字符串转换成整数”要求实现库函数atoi的功能。这就要求我们在平时编程的时候除了能熟练使用库函数，更重要的是要理解库函数的实现原理。

❖ 自以为题目简单的解法

由于不需要考虑大数问题，这道题看起来很简单，可能不少应聘者在看到题目30秒后就能写出如下代码：

```
double Power(double base, int exponent)
{
    double result = 1.0;
    for(int i = 1; i <= exponent; ++i)
```

```

    result *= base;
    return result;
}

```

不过遗憾的是，写得快不一定就能得到面试官的青睐，因为面试官会问如果输入的指数（exponent）小于1（零和负数）的时候怎么办？上面的代码完全没有考虑，只考虑了指数是正数的情况。

❖ 全面但不够高效的解法，我们离 Offer 已经很近了

我们知道，当指数为负数的时候，可以先对指数求绝对值，算出次方的结果之后再取倒数。既然要求倒数，我们很自然地想到有没有可能对0求倒数，如果对0求倒数该怎么办？当底数（base）是零且指数是负数的时候，如果不进行特殊处理，就会出现对0求倒数，从而导致程序运行出错。怎么告诉函数的调用者出现了这种错误？前面提到我们可以采用3种方法：返回值、全局变量和异常。面试的时候可以向面试官阐述每种方法的优缺点，然后一起讨论决定选用哪种方法。

最后需要指出的是，由于0的0次方在数学上是没有意义的，因此无论输出是0还是1都是可以接受的，但这都需要和面试官说清楚，表明我们已经考虑到这个边界值了。

有了这些相对而言已经全面很多的考虑，我们就可以把最初的代码修改如下：

```

bool g_InvalidInput = false;

double Power(double base, int exponent)
{
    g_InvalidInput = false;

    if(equal(base, 0.0) && exponent < 0)
    {
        g_InvalidInput = true;
        return 0.0;
    }

    unsigned int absExponent = (unsigned int)(exponent);
    if(exponent < 0)
        absExponent = (unsigned int)(-exponent);

    double result = PowerWithUnsignedExponent(base, absExponent);
    if(exponent < 0)
        result = 1.0 / result;
}

```

```

        return result;
    }

double PowerWithUnsignedExponent(double base, unsigned int exponent)
{
    double result = 1.0;
    for(int i = 1; i <= exponent; ++i)
        result *= base;

    return result;
}

```

在上述代码中，我们采用全局变量来标识是否出错。如果出错了，则返回的值是 0。但为了区分是出错的时候返回的 0，还是底数为 0 的时候正常运行返回的 0，我们还定义了一个全局变量 `g_InvalidInput`。当出错时，这个变量被设为 `true`，否则为 `false`。这样做的好处是，我们可以把返回值直接传递给其他变量，比如写 `double result = Power(2, 3)`，也可以把函数的返回值直接传递给其他需要 `double` 型参数的函数。但缺点是这个函数的调用者有可能会忘记去检查 `g_InvalidInput` 以判断是否出错，从而留下了安全隐患。由于既有优点也有缺点，因此，我们在写代码之前要和面试官讨论采用哪种出错处理方式最合适。

此时我们已经考虑得很周详了，已经能够达到很多面试官的要求了。但是如果我们碰到的面试官是一个在效率上追求完美的人，那么他有可能会提醒我们函数 `PowerWithUnsignedExponent` 还有更快的办法。

❖ 既全面又高效的解法，确保我们能拿到 Offer

如果输入的指数 `exponent` 为 32，则在函数 `PowerWithUnsignedExponent` 的循环中需要做 31 次乘法。但我们可以换一种思路考虑：我们的目标是求出一个数字的 32 次方，如果我们已经知道了它的 16 次方，那么只要在 16 次方的基础上再平方一次就可以了。而 16 次方是 8 次方的平方。这样以此类推，我们求 32 次方只需要做 5 次乘法：先求平方，在平方的基础上求 4 次方，在 4 次方的基础上求 8 次方，在 8 次方的基础上求 16 次方，最后在 16 次方的基础上求 32 次方。

也就是说，我们可以用如下公式求 a 的 n 次方：

$$a^n = \begin{cases} a^{n/2} \cdot a^{n/2} & n \text{ 为偶数} \\ a^{(n-1)/2} \cdot a^{(n-1)/2} \cdot a & n \text{ 为奇数} \end{cases}$$

这个公式看起来是不是眼熟？我们在介绍用 $O(\log n)$ 时间求斐波那契数列时就讨论过这个公式，这个公式很容易就能通过递归来实现。新的 PowerWithUnsignedExponent 代码如下：

```
double PowerWithUnsignedExponent(double base, unsigned int exponent)
{
    if(exponent == 0)
        return 1;
    if(exponent == 1)
        return base;

    double result = PowerWithUnsignedExponent(base, exponent >> 1);
    result *= result;
    if(exponent & 0x1 == 1)
        result *= base;

    return result;
}
```

最后再提醒一个细节：我们用右移运算符代替了除以 2，用位与运算符代替了求余运算符（%）来判断一个数是奇数还是偶数。位运算的效率比乘除法及求余运算的效率要高很多。既然要优化代码，我们就把优化做到极致。

在面试的时候，我们可以主动提醒面试官注意代码中的两处细节（判断 base 是否等于 0 和用位运算代替乘除法及求余运算），让他知道我们对编程的细节很重视。细节很重要，因为细节决定成败，一两个好的细节说不定就能让面试官下定决心给我们 Offer。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/16_Power



测试用例：

把底数和指数分别设为正数、负数和零。



本题考点：

- 考查应聘者思维的全面性。这个问题本身不难，但能顺利通过的应聘者不是很多。有很多人会忽视底数为 0 而指数为负数时的错误处理。
- 对效率要求比较高的面试官还会考查应聘者快速做乘方的能力。

面试题 17：打印从 1 到最大的 n 位数

题目：输入数字 n ，按顺序打印出从 1 到最大的 n 位十进制数。比如输入 3，则打印出 1、2、3 一直到最大的 3 位数 999。

❖ 跳进面试官陷阱

这道题目看起来很简单。我们看到这个问题之后，最容易想到的办法是先求出最大的 n 位数，然后用一个循环从 1 开始逐个打印。于是我们很容易就能写出如下的代码：

```
void Print1ToMaxOfNDigits_1(int n)
{
    int number = 1;
    int i = 0;
    while(i++ < n)
        number *= 10;

    for(i = 1; i < number; ++i)
        printf("%d\t", i);
}
```

初看之下好像没有问题，但如果仔细分析这个问题，我们就能注意到面试官没有规定 n 的范围。当输入的 n 很大的时候，我们求最大的 n 位数是不是用整型 (int) 或者长整型 (long long) 都会溢出？也就是说我们需要考虑大数问题。这是面试官在这道题里设置的一个大陷阱。

❖ 在字符串上模拟数字加法的解法，绕过陷阱才能拿到 Offer

经过前面的分析，我们很自然地想到解决这个问题需要表达一个大数。最常用也是最容易的方法是用字符串或者数组表达大数。接下来我们用字符串来解决大数问题。

在用字符串表示数字的时候，最直观的方法就是字符串里每个字符都是'0'~'9'之间的某一个字符，用来表示数字中的一位。因为数字最大是 n 位的，因此我们需要一个长度为 $n+1$ 的字符串（字符串中最后一位是结束符号'\0'）。当实际数字不够 n 位的时候，在字符串的前半部分补 0。

首先把字符串中的每一个数字都初始化为'0'，然后每一次为字符串表示的数字加 1，再打印出来。因此，我们只需要做两件事：一是在字符串表达的数字上模拟加法；二是把字符串表达的数字打印出来。

基于上面的分析，我们可以写出如下代码：

```
void Print1ToMaxOfNDigits(int n)
{
    if(n <= 0)
        return;

    char *number = new char[n + 1];
    memset(number, '0', n);
    number[n] = '\0';

    while(!Increment(number))
    {
        PrintNumber(number);
    }

    delete []number;
}
```

在上面的代码中，函数 Increment 实现在表示数字的字符串 number 上增加 1，而函数 PrintNumber 打印出 number。这两个看似简单的函数都暗藏着小小的玄机。

我们需要知道什么时候停止在 number 上增加 1，即什么时候到了最大的 n 位数“999…99”(n 个 9)。一个最简单的办法是在每次递增之后，都调用库函数 strcmp 比较表示数字的字符串 number 和最大的 n 位数“999…99”，如果相等则表示已经到了最大的 n 位数并终止递增。虽然调用 strcmp 很简单，但对于长度为 n 的字符串，它的时间复杂度为 $O(n)$ 。

我们注意到只有对“999…99”加 1 的时候，才会在第一个字符（下标为 0）的基础上产生进位，而其他所有情况都不会在第一个字符上产生进位。因此，当我们发现在加 1 时第一个字符产生了进位，则已经是最大的 n 位数，此时 Increment 返回 true，因此函数 Print1ToMaxOfNDigits 中的 while 循环终止。如何在每一次增加 1 之后快速判断是不是到了最大的 n 位数是本题的一个小陷阱。下面是 Increment 函数的参考代码，它实现了用 $O(1)$ 时间判断是不是已经到了最大的 n 位数。

```
bool Increment(char* number)
{
    bool isOverflow = false;
    int nTakeOver = 0;
    int nLength = strlen(number);
    for(int i = nLength - 1; i >= 0; i--)
    {
        int nSum = number[i] - '0' + nTakeOver;
        if(i == nLength - 1)
            nSum++;
        if(nSum > 9)
            isOverflow = true;
        else
            number[i] = nSum + '0';
        nTakeOver = isOverflow;
    }
}
```

```

        if(nSum >= 10)
    {
        if(i == 0)
            isOverflow = true;
        else
        {
            nSum -= 10;
            nTakeOver = 1;
            number[i] = '0' + nSum;
        }
    }
    else
    {
        number[i] = '0' + nSum;
        break;
    }
}

return isOverflow;
}

```

接下来我们再考虑如何打印用字符串表示的数字。虽然库函数 `printf` 可以很方便地打印出一个字符串，但在本题中调用 `printf` 并不是最合适的解决方案。前面我们提到，当数字不够 n 位的时候，在数字的前面补 0，打印的时候这些补位的 0 不应该打印出来。比如输入 3 的时候，数字 98 用字符串表示成“098”。如果直接打印出 098，就不符合我们的阅读习惯。为此，我们定义了函数 `PrintNumber`，在这个函数里，只有在碰到第一个非 0 的字符之后才开始打印，直至字符串的结尾。能不能按照我们的阅读习惯打印数字是面试官设置的另外一个小陷阱。实现代码如下：

```

void PrintNumber(char* number)
{
    bool isBeginning0 = true;
    int nLength = strlen(number);

    for(int i = 0; i < nLength; ++ i)
    {
        if(isBeginning0 && number[i] != '0')
            isBeginning0 = false;

        if(!isBeginning0)
        {
            printf("%c", number[i]);
        }
    }

    printf("\t");
}

```

❖ 把问题转换成数字排列的解法，递归让代码更简洁

上述思路虽然比较直观，但由于模拟了整数的加法，代码有点长。要在面试短短几十分钟时间里完整、正确地写出这么长的代码，对很多应聘者而言不是一件容易的事情。接下来我们换一种思路来考虑这个问题。如果我们在数字前面补 0，就会发现 n 位所有十进制数其实就是 n 个从 0 到 9 的全排列。也就是说，我们把数字的每一位都从 0 到 9 排列一遍，就得到了所有的十进制数。只是在打印的时候，排在前面的 0 不打印出来罢了。

全排列用递归很容易表达，数字的每一位都可能是 0~9 中的一个数，然后设置下一位。递归结束的条件是我们已经设置了数字的最后一位。

```
void Print1ToMaxOfNDigits (int n)
{
    if(n <= 0)
        return;

    char* number = new char[n + 1];
    number[n] = '\0';

    for(int i = 0; i < 10; ++i)
    {
        number[0] = i + '0';
        Print1ToMaxOfNDigitsRecursively(number, n, 0);
    }

    delete[] number;
}

void Print1ToMaxOfNDigitsRecursively(char* number, int length, int index)
{
    if(index == length - 1)
    {
        PrintNumber(number);
        return;
    }

    for(int i = 0; i < 10; ++i)
    {
        number[index + 1] = i + '0';
        Print1ToMaxOfNDigitsRecursively(number, length, index + 1);
    }
}
```

函数 PrintNumber 和前面第二种思路中的一样，这里就不再重复了。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/17_Print1ToMaxOfNDigits



测试用例：

- 功能测试（输入1、2、3……）。
- 特殊输入测试（输入-1、0）。



本题考点：

- 考查应聘者解决大数问题的能力。面试官出这道题目时候，他期望应聘者能意识到这是一个大数问题，同时还期待应聘者能定义合适的数据表示方式来解决大数问题。
- 如果应聘者采用第一种思路，即在数字上加1逐个打印的思路，则面试官会关注他判断是否已经到了最大的n位数时采用的方法。应聘者要注意到不同方法的时间效率相差很大。
- 如果应聘者采用第二种思路，则面试官还将考查他用递归方法解决问题的能力。
- 面试官还将关注应聘者打印数字时会不会打印出位于数字前面的0。这里能体现出应聘者在设计开发软件时是不是会考虑用户的使用习惯。通常我们的软件设计和开发需要符合大部分用户的人机交互习惯。



本题扩展：

在前面的代码中，我们用一个char型字符表示十进制数字的一位。8bit的char型字符最多能表示256个字符，而十进制数字只有0~9的10个数字。因此，用char型字符来表示十进制数字并没有充分利用内存，有一些浪费。有没有更高效的方式来表示大数？



相关题目：

定义一个函数，在该函数中可以实现任意两个整数的加法。由于没有限定输入两个数的大小范围，我们也要把它当作大数问题来处理。在前面的代码的第一种思路中，实现了在字符串表示的数字上加 1 的功能，我们可以参考这种思路实现两个数字的相加功能。另外还有一个需要注意的问题：如果输入的数字中有负数，那么我们应该怎么处理？



面试小提示：

如果面试题是关于 n 位的整数并且没有限定 n 的取值范围，或者输入任意大小的整数，那么这道题目很有可能是需要考虑大数问题的。字符串是一种简单、有效地表示大数的方法。

面试题 18：删除链表的节点

题目一：在 $O(1)$ 时间内删除链表节点。

给定单向链表的头指针和一个节点指针，定义一个函数在 $O(1)$ 时间内删除该节点。链表节点与函数的定义如下：

```
struct ListNode
{
    int         m_nValue;
    ListNode*  m_pNext;
};

void DeleteNode(ListNode** pListHead, ListNode* pToDelete);
```

在单向链表中删除一个节点，常规的做法无疑是从链表的头节点开始，顺序遍历查找要删除的节点，并在链表中删除该节点。

比如在如图 3.3 (a) 所示的链表中，我们想删除节点 i，可以从链表的头节点 a 开始顺序遍历，发现节点 h 的 m_pNext 指向要删除的节点 i，于是我们可以把节点 h 的 m_pNext 指向 i 的下一个节点，即节点 j。指针调整之后，我们就可以安全地删除节点 i 并保证链表没有断开，如图 3.3 (b) 所示。这种思路上由于需要顺序查找，时间复杂度自然就是 $O(n)$ 了。

之所以需要从头开始查找，是因为我们需要得到将被删除的节点的前一个节点。在单向链表中，节点中没有指向前一个节点的指针，所以只好从链表的头节点开始顺序查找。

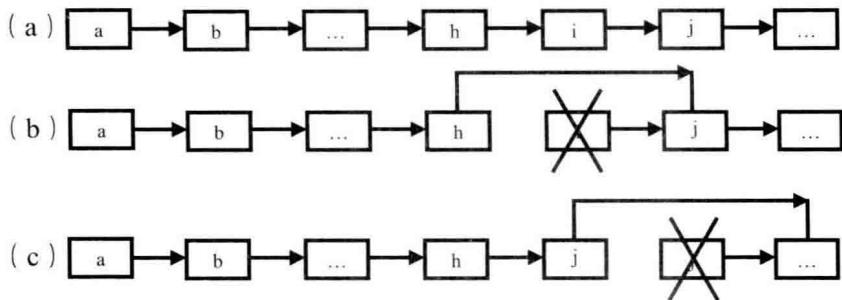


图 3.3 在链表中删除一个节点的两种方法

注：(a) 一个链表。(b) 在删除节点 i 之前，先从链表的头节点开始遍历到 i 前面的一个节点 h，把 h 的 `m_pNext` 指向 i 的下一个节点 j，再删除节点 i。(c) 把节点 j 的内容复制覆盖节点 i，接下来再把节点 i 的 `m_pNext` 指向 j 的下一个节点，再删除节点 j。这种方法不用遍历链表上节点 i 前面的节点。

那是不是一定需要得到被删除的节点的前一个节点呢？答案是否定的。我们可以很方便地得到要删除的节点的下一个节点。如果我们把下一个节点的内容复制到需要删除的节点上覆盖原有的内容，再把下一个节点删除，那是不是就相当于把当前需要删除的节点删除了？

我们还是以前面的例子来分析这种思路。我们要删除节点 i，先把 i 的下一个节点 j 的内容复制到 i，然后把 i 的指针指向节点 j 的下一个节点。此时再删除节点 j，其效果刚好是把节点 i 删除了，如图 3.3 (c) 所示。

上述思路还有一个问题：如果要删除的节点位于链表的尾部，那么它就没有下一个节点，怎么办？我们仍然从链表的头节点开始，顺序遍历得到该节点的前序节点，并完成删除操作。

最后需要注意的是，如果链表中只有一个节点，而我们又要删除链表的头节点（也是尾节点），那么，此时我们在删除节点之后，还需要把链表的头节点设置为 `nullptr`。

有了这些思路，我们就可以动手写代码了。下面是这种思路的参考代码：

```
void DeleteNode(ListNode** pListHead, ListNode* pToDelete)
{
    if(!pListHead || !pToDelete)
        return;
```

```

// 要删除的节点不是尾节点
if(pToBeDeleted->m_pNext != nullptr)
{
    ListNode* pNext = pToBeDeleted->m_pNext;
    pToBeDeleted->m_nValue = pNext->m_nValue;
    pToBeDeleted->m_pNext = pNext->m_pNext;

    delete pNext;
    pNext = nullptr;
}

// 链表只有一个节点，删除头节点（也是尾节点）
else if(*pListHead == pToBeDeleted)
{
    delete pToBeDeleted;
    pToBeDeleted = nullptr;
    *pListHead = nullptr;
}

// 链表中有多个节点，删除尾节点
else
{
    ListNode* pNode = *pListHead;
    while(pNode->m_pNext != pToBeDeleted)
    {
        pNode = pNode->m_pNext;
    }

    pNode->m_pNext = nullptr;
    delete pToBeDeleted;
    pToBeDeleted = nullptr;
}
}

```

接下来我们分析这种思路的时间复杂度。对于 $n-1$ 个非尾节点而言，我们可以在 $O(1)$ 时间内把下一个节点的内存复制覆盖要删除的节点，并删除下一个节点；对于尾节点而言，由于仍然需要顺序查找，时间复杂度是 $O(n)$ 。因此，总的平均时间复杂度是 $[(n-1) \times O(1) + O(n)]/n$ ，结果还是 $O(1)$ ，符合面试官的要求。

值得注意的是，上述代码仍然不是完美的代码，因为它基于一个假设：要删除的节点的确在链表中。我们需要 $O(n)$ 的时间才能判断链表中是否包含某一节点。受到 $O(1)$ 时间的限制，我们不得不把确保节点在链表中的责任推给了函数 DeleteNode 的调用者。在面试的时候，我们可以和面试官讨论这个假设，这样面试官就会觉得我们考虑问题非常全面。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/18_01_DeleteNodeInList



测试用例：

- 功能测试（从有多个节点的链表的中间删除一个节点；从有多个节点的链表中删除头节点；从有多个节点的链表中删除尾节点；从只有一个节点的链表中删除唯一的节点）。
- 特殊输入测试（指向链表头节点的为 `nullptr` 指针；指向要删除节点的为 `nullptr` 指针）。



本题考点：

- 考查应聘者对链表的编程能力。
- 考查应聘者的创新思维能力。这道题要求应聘者打破常规的思维模式。当我们想删除一个节点时，并不一定要删除这个节点本身。可以先把下一个节点的内容复制出来覆盖被删除节点的内容，然后把下一个节点删除。这种思路不是很容易想到的。
- 考查应聘者思维的全面性。即使应聘者想到删除下一个节点这个办法，也未必能通过这轮面试。应聘者要全面考虑删除的节点位于链表的尾部及输入的链表只有一个节点这些特殊情况。

题目二：删除链表中重复的节点。

在一个排序的链表中，如何删除重复的节点？例如，在图 3.4 (a) 中重复的节点被删除之后，链表如图 3.4 (b) 所示。

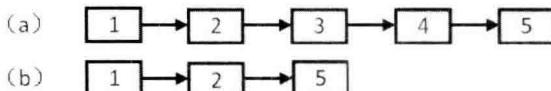


图 3.4 删除链表中重复的节点

注：(a) 一个有 7 个节点的链表；(b) 当重复的节点被删除之后，链表中只剩下 3 个节点。

解决这个问题的第一步是确定删除函数的参数。当然，这个函数需要输入待删除链表的头节点。头节点可能与后面的节点重复，也就是说头节

点也可能被删除，因此删除函数应该声明为 void deleteDuplication(ListNode** pHead)，而不是 void deleteDuplication(ListNode* pHead)。

接下来我们从头遍历整个链表。如果当前节点（代码中的 pNode）的值与下一个节点的值相同，那么它们就是重复的节点，都可以被删除。为了保证删除之后的链表仍然是相连的，我们要把当前节点的前一个节点（代码中的 pPreNode）和后面值比当前节点的值大的节点相连。我们要确保 pPreNode 始终与下一个没有重复的节点连接在一起。

我们以图 3.4 中的链表为例来分析删除重复节点的过程。当我们遍历到第一个值为 3 的节点的时候，pPreNode 指向值为 2 的节点。由于接下来的节点的值还是 3，这两个节点应该被删除，因此 pPreNode 就和第一个值为 4 的节点相连。接下来由于值为 4 的两个节点也重复了，还是会被删除，所以 pPreNode 最终会和值为 5 的节点相连。

上述删除重复节点的过程可以用如下代码实现：

```
void DeleteDuplication(ListNode** pHead)
{
    if(pHead == nullptr || *pHead == nullptr)
        return;

    ListNode* pPreNode = nullptr;
    ListNode* pNode = *pHead;
    while(pNode != nullptr)
    {
        ListNode *pNext = pNode->m_pNext;
        bool needDelete = false;
        if(pNext != nullptr && pNext->m_nValue == pNode->m_nValue)
            needDelete = true;

        if(!needDelete)
        {
            pPreNode = pNode;
            pNode = pNode->m_pNext;
        }
        else
        {
            int value = pNode->m_nValue;
            ListNode* pToBeDel = pNode;
            while(pToBeDel != nullptr && pToBeDel->m_nValue == value)
            {
                pNext = pToBeDel->m_pNext;

                delete pToBeDel;
                pToBeDel = nullptr;

                pToBeDel = pNext;
            }
        }
    }
}
```

```

        if(pPreNode == nullptr)
            *pHead = pNext;
        else
            pPreNode->m_pNext = pNext;
        pNode = pNext;
    }
}
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/18_02_DeleteDuplicatedNode



测试用例：

- 功能测试（重复的节点位于链表的头部/中间/尾部；链表中没有重复的节点）。
- 特殊输入测试（指向链表头节点的为 `nullptr` 指针；链表中所有节点都是重复的）。



本题考点：

- 考查应聘者对链表的编程能力。
- 考查应聘者思维的全面性。应聘者要全面考虑重复节点所处的位置，以及删除重复节点之后的结果。

面试题 19：正则表达式匹配

题目：请实现一个函数用来匹配包含'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（含0次）。在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但与"aa.a"和"ab*a"均不匹配。

每次从字符串里拿出一个字符和模式中的字符去匹配。先来分析如何匹配一个字符。如果模式中的字符 `ch` 是'.'，那么它可以匹配字符串中的任意字符。如果模式中的字符 `ch` 不是'.'，而且字符串中的字符也是 `ch`，那么

它们相互匹配。当字符串中的字符和模式中的字符相匹配时，接着匹配后面的字符。

相对而言，当模式中的第二个字符不是'*'时，问题要简单很多。如果字符串中的第一个字符和模式中的第一个字符相匹配，那么在字符串和模式上都向后移动一个字符，然后匹配剩余的字符串和模式。如果字符串中的第一个字符和模式中的第一个字符不相匹配，则直接返回 false。

当模式中的第二个字符是'*'时，问题要复杂一些，因为可能有多种不同的匹配方式。一种选择是在模式上向后移动两个字符。这相当于'*'和它前面的字符被忽略了，因为'*'可以匹配字符串中的 0 个字符。如果模式中的第一个字符和字符串中的第一个字符相匹配，则在字符串上向后移动一个字符，而在模式上有两种选择：可以在模式上向后移动两个字符，也可以保持模式不变。

如图 3.5 所示，当匹配进入状态 2 并且字符串的字符是'a'时，我们有两种选择：可以进入状态 3（在模式上向后移动两个字符），也可以回到状态 2（模式保持不变）。

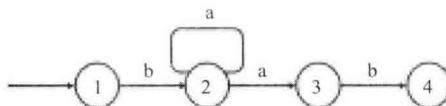


图 3.5 模式 ba^*ab 的非确定有限状态机

根据上述分析，可以写出如下代码：

```

bool match(char* str, char* pattern)
{
    if(str == nullptr || pattern == nullptr)
        return false;
    return matchCore(str, pattern);
}

bool matchCore(char* str, char* pattern)
{
    if(*str == '\0' && *pattern == '\0')
        return true;
    if(*str != '\0' && *pattern == '\0')
        return false;
    if(*(pattern + 1) == '*')
    {
        if(*pattern == *str || (*pattern == '.' && *str != '\0'))

```

```

        // move on the next state
    return matchCore(str + 1, pattern + 2)
    // stay on the current state
    || matchCore(str + 1, pattern)
    // ignore a '*'
    || matchCore(str, pattern + 2);
else
    // ignore a '*'
    return matchCore(str, pattern + 2);
}

if(*str == *pattern || (*pattern == '.' && *str != '0'))
    return matchCore(str + 1, pattern + 1);

return false;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/19_RegularExpressionsMatching



测试用例：

- 功能测试（模式字符串里包含普通字符、'.'、'*'；模式字符串和输入字符串匹配/不匹配）。
- 特殊输入测试（输入字符串和模式字符串是 nullptr、空字符串）。



本题考点：

- 考查应聘者对字符串的编程能力。
- 考查应聘者对正则表达式的理解。
- 考查应聘者思维的全面性。应聘者要全面考虑普通字符、'.'和'*'并分别分析它们的匹配模式。在应聘者写完代码之后，面试官会要求他/她测试自己的代码。这时候应聘者要充分考虑普通字符、'.'和'*'三者的排列组合，尽量完备地想出所有可能的测试用例。

面试题 20：表示数值的字符串

题目：请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100"、"5e2"、"-123"、"3.1416"及"-1E-16"都表示数值，但"12e"、"1a3.14"、"1.2.3"、"+-5"及"12e+5.4"都不是。

表示数值的字符串遵循模式 A[.B][e|EC]或者.B[e|EC]，其中 A 为数值的整数部分，B 紧跟着小数点为数值的小数部分，C 紧跟着'e'或者'E'为数值的指数部分。在小数里可能没有数值的整数部分。例如，小数.123 等于 0.123。因此 A 部分不是必需的。如果一个数没有整数部分，那么它的小数部分不能为空。

上述 A 和 C 都是可能以 '+' 或者 '-' 开头的 0~9 的数位串；B 也是 0~9 的数位串，但前面不能有正负号。

以表示数值的字符串"123.45e+6"为例，“123”是它的整数部分 A，“45”是它的小数部分 B，“+6”是它的指数部分 C。

判断一个字符串是否符合上述模式时，首先尽可能多地扫描 0~9 的数位（有可能在起始处有 '+' 或者 '-'），也就是前面模式中表示数值整数的 A 部分。如果遇到小数点 '!'，则开始扫描表示数值小数部分的 B 部分。如果遇到 'e' 或者 'E'，则开始扫描表示数值指数的 C 部分。

整个过程可以用如下代码实现：

```
// 数字的格式可以用 A[.B][e|EC]或者.B[e|EC]表示，其中 A 和 C 都是
// 整数（可以有正负号，也可以没有），而 B 是一个无符号整数
bool isNumeric(const char* str)
{
    if(str == nullptr)
        return false;

    bool numeric = scanInteger(&str);

    // 如果出现 '!'，则接下来是数字的小数部分
    if(*str == '!')
    {
        ++str;

        // 下面一行代码用 || 的原因：
        // 1. 小数可以没有整数部分，如.123 等于 0.123;
        // 2. 小数点后面可以没有数字，如 233. 等于 233.0;
        // 3. 当然，小数点前面和后面可以都有数字，如 233.666
        numeric = scanUnsignedInteger(&str) || numeric;
    }
}
```

```

// 如果出现'e'或者'E'，则接下来是数字的指数部分
if(*str == 'e' || *str == 'E')
{
    ++str;
    // 下面一行代码用&&的原因：
    // 1. 当 e 或 E 前面没有数字时，整个字符串不能表示数字，如.e1、e1；
    // 2. 当 e 或 E 后面没有整数时，整个字符串不能表示数字，如 12e、12e+5.4
    numeric = numeric && scanInteger(&str);
}

return numeric && *str == '\0';
}

```

函数 `scanUnsignedInteger` 用来扫描字符串中 0~9 的数位（类似于一个无符号整数），可以用来匹配前面数值模式中的 B 部分。

```

bool scanUnsignedInteger(const char** str)
{
    const char* before = *str;
    while(**str != '\0' && **str >= '0' && **str <= '9')
        ++(*str);

    // 当 str 中存在若干 0~9 的数字时，返回 true
    return *str > before;
}

```

函数 `scanInteger` 扫描可能以表示正负的 '+' 或者 '-' 为起始的 0~9 的数位（类似于一个可能带正负符号的整数），用来匹配前面数值模式中的 A 和 C 部分。

```

bool scanInteger(const char** str)
{
    if(**str == '+' || **str == '-')
        ++(*str);
    return scanUnsignedInteger(str);
}

```



源代码：

本题完整的源代码（含单元测试用例）：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/20_NumericStrings



测试用例：

- 功能测试（正数或者负数；包含或者不包含整数部分的数值；包含或者不包含小数部分的数值；包含或者不包含指数部分的数值；各种不能表达有效数值的字符串）。
- 特殊输入测试（输入字符串和模式字符串是 nullptr、空字符串）。



本题考点：

- 考查应聘者对字符串的编程能力。
- 考查应聘者分析问题的能力。面试官希望应聘者能够从不同类型的数值中分析出规律。
- 考查应聘者思维的全面性。应聘者要全面考虑数值整数、小数、指数部分的特点，比如哪些部分可以出现正负号，而哪些部分不能出现。在应聘者写完代码之后，面试官会期待应聘者能够完备地测试用例来验证自己的代码。

面试题 21：调整数组顺序使奇数位于偶数前面

题目：输入一个整数数组，实现一个函数来调整该数组中数字的顺序，使得所有奇数位于数组的前半部分，所有偶数位于数组的后半部分。

如果不考虑时间复杂度，则最简单的思路应该是从头扫描这个数组，每碰到一个偶数，拿出这个数字，并把位于这个数字后面的所有数字往前挪动一位。挪完之后在数组的末尾有一个空位，这时把该偶数放入这个空位。由于每碰到一个偶数就需要移动 $O(n)$ 个数字，因此总的时间复杂度是 $O(n^2)$ 。但是，这种方法不能让面试官满意。不过如果我们在听到题目之后能马上说出这种解法，那么面试官至少会觉得我们的思维非常敏捷。

❖ 只完成基本功能的解法，仅适用于初级程序员

这道题目要求把奇数放在数组的前半部分，偶数放在数组的后半部分，因此所有的奇数应该位于偶数的前面。也就是说，我们在扫描这个数组的时候，如果发现有偶数出现在奇数的前面，则交换它们的顺序，交换之后就符合要求了。

因此，我们可以维护两个指针：第一个指针初始化时指向数组的第一个数字，它只向后移动；第二个指针初始化时指向数组的最后一个数字，它只向前移动。在两个指针相遇之前，第一个指针总是位于第二个指针的前面。如果第一个指针指向的数字是偶数，并且第二个指针指向的数字是奇数，则交换这两个数字。

下面以一个具体的例子，如输入数组{1, 2, 3, 4, 5}来分析这种思路。在初始化时，把第一个指针指向数组第一个数字1，而把第二个指针指向最后一个数字5，如图3.6(a)所示。第一个指针指向的数字1是一个奇数，不需要处理，我们把第一个指针向后移动，直到碰到一个偶数2。此时第二个指针已经指向了奇数，因此不需要移动。此时两个指针指向的位置如图3.6(b)所示。这时候我们发现偶数2位于奇数5的前面，符合交换条件，于是交换这两个指针指向的数字，如图3.6(c)所示。

接下来我们继续向后移动第一个指针，直到碰到下一个偶数4，并向前移动第二个指针，直到碰到第一个奇数3，如图3.6(d)所示。我们发现第二个指针已经在第一个指针的前面了，表示所有的奇数都已经在偶数的前面了。此时的数组是{1, 5, 3, 4, 2}，的确是奇数位于数组的前半部分而偶数位于数组的后半部分。

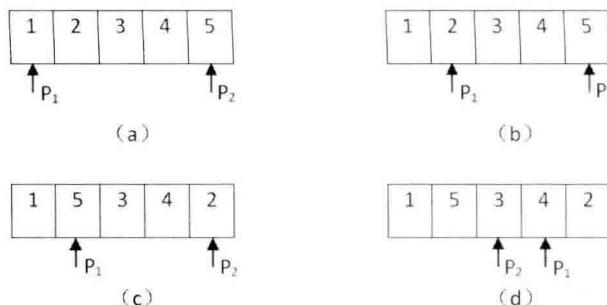


图3.6 调整数组{1, 2, 3, 4, 5}，使得奇数位于偶数前面的过程

注：(a) 把第一个指针指向数组的第一个数字，第二个指针指向最后一个数字。(b) 向后移动第一个指针直至它指向偶数2，此时第二个指针指向奇数5，不需要移动。(c) 交换两个指针指向的数字。(d) 向后移动第一个指针直至它指向偶数4，向前移动第二个指针直至它指向奇数3。由于第二个指针移到了第一个指针的前面，表明所有的奇数都位于偶数的前面。

基于这个分析，我们可以写出如下代码：

```

void ReorderOddEven(int *pData, unsigned int length)
{
    if(pData == nullptr || length == 0)
        return;

    int *pBegin = pData;
    int *pEnd = pData + length - 1;

    while(pBegin < pEnd)
    {
        // 向后移动 pBegin, 直到它指向偶数
        while(pBegin < pEnd && (*pBegin & 0x1) != 0)
            pBegin++;

        // 向前移动 pEnd, 直到它指向奇数
        while(pBegin < pEnd && (*pEnd & 0x1) == 0)
            pEnd--;

        if(pBegin < pEnd)
        {
            int temp = *pBegin;
            *pBegin = *pEnd;
            *pEnd = temp;
        }
    }
}

```

❖ 考虑可扩展的解法，能秒杀 Offer

如果面试应届毕业生或者工作时间不长的程序员，则面试官会满意前面的代码。但如果应聘者申请的是资深的开发职位，那么面试官可能会接着问几个问题。

面试官：如果把题目改成把数组中的数按照大小分为两部分，所有负数都在非负数的前面，该怎么做？

应聘者：这很简单，可以重新定义一个函数。在新的函数里，只要修改第二个和第三个 while 循环中的判断条件就行了。

面试官：如果再把题目改改，变成把数组中的数分为两部分，能被 3 整除的数都在不能被 3 整除的数的前面。怎么办？

应聘者：我们还是可以定义一个新的函数。在这个函数中……

面试官：（打断应聘者的话）难道就没有更好的办法？

这时候应聘者要反应过来，面试官期待我们提供的不仅仅是解决一个问题的办法，而是解决一系列同类型问题的通用办法。这就是面试官在考

查我们对扩展性的理解，即希望我们能够给出一种模式，在这种模式下能够很方便地把已有的解决方案扩展到同类型的问题上去。

回到面试官新提出的两个问题上来。我们发现，要解决这两个新的问题，其实只需要修改函数 ReorderOddEven 中的两处判断的标准，而大的逻辑框架完全不需要改动。因此我们可以把这个逻辑框架抽象出来，而把判断的标准变成一个函数指针，也就是用一个单独的函数来判断数字是不是符合标准。这样我们就把整个函数解耦成两部分：一是判断数字应该在数组前半部分还是后半部分的标准；二是拆分数组的操作。于是我们可以写出下面的代码：

```
void Reorder(int *pData, unsigned int length, bool (*func)(int))
{
    if(pData == nullptr || length == 0)
        return;

    int *pBegin = pData;
    int *pEnd = pData + length - 1;

    while(pBegin < pEnd)
    {
        while(pBegin < pEnd && !func(*pBegin))
            pBegin++;

        while(pBegin < pEnd && func(*pEnd))
            pEnd--;
    }

    if(pBegin < pEnd)
    {
        int temp = *pBegin;
        *pBegin = *pEnd;
        *pEnd = temp;
    }
}

bool isEven(int n)
{
    return (n & 1) == 0;
}
```

在上面的代码中，函数 Reorder 根据 func 的标准把数组 pData 分成两部分；而函数 isEven 则是一个具体的标准，即判断一个数是不是偶数。有了这两个函数，我们就可以很方便地把数组中的所有奇数移到偶数的前面。实现代码如下：

```
void ReorderOddEven(int *pData, unsigned int length)
{
```

```

    Reorder(pData, length, isEven);
}

```

如果把问题改成将数组中的负数移到非负数的前面，或者把能被 3 整除的数移到不能被 3 整除的数的前面，都只需定义新的函数来确定分组的标准，而函数 Reorder 不需要进行任何改动。也就是说，解耦的好处就是提高了代码的重用性，为功能扩展提供了便利。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/21_ReorderArray



测试用例：

- 功能测试（输入数组中的奇数、偶数交替出现；输入的数组中所有偶数都出现在奇数的前面；输入的数组中所有奇数都出现在偶数的前面）。
- 特殊输入测试（输入 nullptr 指针；输入的数组只包含一个数字）。



本题考点：

- 考查应聘者的快速思维能力。要在短时间内按照要求把数组分隔成两部分不是一件容易的事情，需要较快的思维能力。
- 对于已经工作了几年的应聘者，面试官还将考查其对扩展性的理解，要求应聘者写出的代码具有可重用性。

3.4 代码的鲁棒性

鲁棒是英文 Robust 的音译，有时也翻译成健壮性。所谓的鲁棒性是指程序能够判断输入是否合乎规范要求，并对不符合要求的输入予以合理的处理。

容错性是鲁棒性的一个重要体现。不鲁棒的软件在发生异常事件的时候，比如用户输入错误的用户名、试图打开的文件不存在或者网络不能连接，就会出现不可预见的诡异行为，或者干脆整个软件崩溃。这样的软件对于用户而言，不亚于一场灾难。

由于鲁棒性对软件开发非常重要，所以面试官在招聘的时候对应聘者写出的代码是否鲁棒也非常关注。提高代码的鲁棒性的有效途径是进行防御性编程。防御性编程是一种编程习惯，是指预见在什么地方可能会出现问题，并为这些可能出现的问题制定处理方式。比如，当试图打开文件时发现文件不存在，可以提示用户检查文件名和路径；当服务器连接不上时，可以试图连接备用服务器等。这样，当异常情况发生时，软件的行为也尽在我们的掌握之中，而不至于出现不可预见的事情。

在面试时，最简单也最实用的防御性编程就是在函数入口添加代码以验证用户输入是否符合要求。通常面试要求的是写一两个函数，我们需要格外关注这些函数的输入参数。如果输入的是一个指针，那么指针是空指针怎么办？如果输入的是一个字符串，那么字符串的内容为空怎么办？如果能把这些问题都提前考虑到，并进行相应的处理，那么面试官就会觉得我们有防御性编程的习惯，能够写出鲁棒的软件。

当然，并不是所有与鲁棒性相关的问题都只是检查输入的参数这么简单。我们看到问题的时候，要多问几个“如果不……那么……”这样的问题。比如面试题22“链表中倒数第 k 个节点”，这里就隐含着一个条件，即链表中节点的个数大于 k 。我们就要问：如果链表中节点的数目不是大于 k 个，那么代码会出现什么问题？这样的思考方式能够帮助我们发现潜在的问题并提前解决问题。这比让面试官发现问题之后我们再去慌忙分析代码、查找问题的根源要好得多。

面试题 22：链表中倒数第 k 个节点

题目：输入一个链表，输出该链表中倒数第 k 个节点。为了符合大多数人的习惯，本题从1开始计数，即链表的尾节点是倒数第1个节点。例如，一个链表有6个节点，从头节点开始，它们的值依次是1、2、3、4、5、6。这个链表的倒数第3个节点是值为4的节点。链表节点定义如下：

```
struct ListNode
{
```

```

int      m_nValue;
ListNode* m_pNext;
};

```

为了得到倒数第 k 个节点，很自然的想法是先走到链表的尾端，再从尾端回溯 k 步。可是我们从链表节点的定义可以看出，本题中的链表是单向链表，单向链表的节点只有从前往后的指针而没有从后往前的指针，因此这种思路行不通。

既然不能从尾节点开始遍历这个链表，我们还是把思路回到头节点上来。假设整个链表有 n 个节点，那么倒数第 k 个节点就是从头节点开始的第 $n-k+1$ 个节点。如果我们能够得到链表中节点的个数 n ，那么只要从头节点开始往后走 $n-k+1$ 步就可以了。如何得到节点数 n ？这个不难，只需要从头开始遍历链表，每经过一个节点，计数器加 1 就行了。

也就是说我们需要遍历链表两次，第一次统计出链表中节点的个数，第二次就能找到倒数第 k 个节点。但是当我们把这种思路解释给面试官之后，他会告诉我们他期待的解法只需要遍历链表一次。

为了实现只遍历链表一次就能找到倒数第 k 个节点，我们可以定义两个指针。第一个指针从链表的头指针开始遍历向前走 $k-1$ 步，第二个指针保持不动；从第 k 步开始，第二个指针也开始从链表的头指针开始遍历。由于两个指针的距离保持在 $k-1$ ，当第一个（走在前面的）指针到达链表的尾节点时，第二个（走在后面的）指针正好指向倒数第 k 个节点。

下面以在有 6 个节点的链表中找倒数第 3 个节点为例分析这种思路的过程。首先用第一个指针从头节点开始向前走两 ($2=3-1$) 步到达第 3 个节点，如图 3.7 (a) 所示。接着把第二个指针初始化指向链表的第一个节点，如图 3.7 (b) 所示。最后让两个指针同时向前遍历，当第一个指针到达链表的尾节点时，第二个指针指向的刚好就是倒数第 3 个节点，如图 3.7 (c) 所示。

想清楚这种思路之后，很多人很快就能写出如下代码：

```

ListNode* FindKthToTail(ListNode* pListHead, unsigned int k)
{
    ListNode *pAhead = pListHead;
    ListNode *pBehind = nullptr;

    for(unsigned int i = 0; i < k - 1; ++ i)
    {
        pAhead = pAhead->m_pNext;
    }
}

```

```

pBehind = pListHead;

while(pAhead->m_pNext != nullptr)
{
    pAhead = pAhead->m_pNext;
    pBehind = pBehind->m_pNext;
}

return pBehind;
}

```

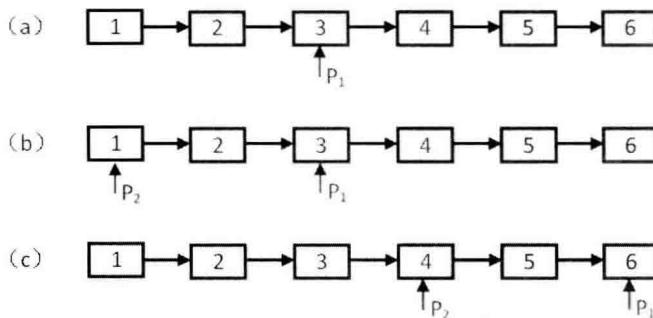


图 3.7 在有 6 个节点的链表上找倒数第 3 个节点的过程

注: (a) 第一个指针在链表上走两步。(b) 把第二个指针指向链表的头节点。(c) 两个指针一同沿着链表向前走。当第一个指针指向链表的尾节点时, 第二个指针指向倒数第 3 个节点。

有不少人在面试之前从网上看到过用两个指针遍历的思路来解这道题, 因此听到面试官问这道题, 他们心中一阵窃喜, 很快就能写出代码。可是几天之后他们等来的不是 Offer, 而是拒信, 于是百思不得其解。其实原因很简单, 就是自己写的代码不够鲁棒。以上面的代码为例, 面试官可以找出 3 种办法让这段代码崩溃。

(1) 输入的 $pListHead$ 为空指针。由于代码会试图访问空指针指向的内存, 从而造成程序崩溃。

(2) 输入的以 $pListHead$ 为头节点的链表的节点总数少于 k 。由于在 for 循环中会在链表上向前走 $k-1$ 步, 仍然会由于空指针而造成程序崩溃。

(3) 输入的参数 k 为 0。由于 k 是一个无符号整数, 那么在 for 循环中 $k-1$ 得到的将不是 -1, 而是 4294967295 (无符号的 0xFFFFFFFF)。因此, for 循环执行的次数远远超出我们的预计, 同样也会造成程序崩溃。

这么简单的代码却存在 3 个潜在崩溃的风险，我们可以想象当面试官看到这样的代码时会有什么样的心情，最终他给出的是拒信而不是 Offer 虽是意料之外，但也在情理之中。



面试小提示：

面试过程中写代码要特别注意鲁棒性。如果写出的代码存在多处崩溃的风险，那么我们很有可能和 Offer 失之交臂。

针对前面指出的 3 个问题，我们要分别处理。如果输入的链表头指针为 `nullptr`，那么整个链表为空，此时查找倒数第 k 个节点自然应该返回 `nullptr`。如果输入的 k 是 0，也就是试图查找倒数第 0 个节点，由于我们计数是从 1 开始的，因此输入 0 没有实际意义，也可以返回 `nullptr`。如果链表的节点数少于 k ，那么在 `for` 循环中遍历链表可能会出现指向 `nullptr` 的 `m_pNext`，因此我们在 `for` 循环中应该加一个 `if` 判断。修改之后的代码如下：

```
ListNode* FindKthToTail(ListNode* pListHead, unsigned int k)
{
    if(pListHead == nullptr || k == 0)
        return nullptr;

    ListNode *pAhead = pListHead;
    ListNode *pBehind = nullptr;

    for(unsigned int i = 0; i < k - 1; ++ i)
    {
        if(pAhead->m_pNext != nullptr)
            pAhead = pAhead->m_pNext;
        else
        {
            return nullptr;
        }
    }

    pBehind = pListHead;
    while(pAhead->m_pNext != nullptr)
    {
        pAhead = pAhead->m_pNext;
        pBehind = pBehind->m_pNext;
    }

    return pBehind;
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/22_KthNodeFromEnd



测试用例：

- 功能测试(第 k 个节点在链表的中间; 第 k 个节点是链表的头节点; 第 k 个节点是链表的尾节点)。
- 特殊输入测试(链表头节点为 `nullptr` 指针; 链表的节点总数少于 k ; k 等于 0)。



本题考点：

- 考查应聘者对链表的理解。
- 考查应聘者所写代码的鲁棒性。鲁棒性是解决这道题的关键所在。如果应聘者写出的代码有着多处崩溃的潜在风险，那么他是很难通过这轮面试的。



相关题目：

求链表的中间节点。如果链表中的节点总数为奇数，则返回中间节点；如果节点总数是偶数，则返回中间两个节点的任意一个。为了解决这个问题，我们也可以定义两个指针，同时从链表的头节点出发，一个指针一次走一步，另一个指针一次走两步。当走得快的指针走到链表的末尾时，走得慢的指针正好在链表的中间。



举一反三：

当我们用一个指针遍历链表不能解决问题的时候，可以尝试用两个指针来遍历链表。可以让其中一个指针遍历的速度快一些（比如一次在链表上走两步），或者让它先在链表上走若干步。

面试题 23：链表中环的入口节点

题目：如果一个链表中包含环，如何找出环的入口节点？例如，在如图 3.8 所示的链表中，环的入口节点是节点 3。

解决这个问题的第一步是如何确定一个链表中包含环。受到面试题 22 的启发，我们可以用两个指针来解决这个问题。和前面的问题一样，定义两个指针，同时从链表的头节点出发，一个指针一次走一步，另一个指针一次走两步。如果走得快的指针追上了走得慢的指针，那么链表就包含环；如果走得快的指针走到了链表的末尾 (`m_pNext` 指向 `NULL`) 都没有追上第一个指针，那么链表就不包含环。

第二步是如何找到环的入口。我们还是可以用两个指针来解决这个问题。先定义两个指针 P_1 和 P_2 指向链表的头节点。如果链表中的环有 n 个节点，则指针 P_1 先在链表上向前移动 n 步，然后两个指针以相同的速度向前移动。当第二个指针指向环的入口节点时，第一个指针已经围绕着环走了一圈，又回到了入口节点。



图 3.8 节点 3 是链表中环的入口节点

以图 3.8 为例分析两个指针的移动规律。指针 P_1 和 P_2 在初始化时都指向链表的头节点，如图 3.9 (a) 所示。由于环中有 4 个节点，所以指针 P_1 先在链表上向前移动 4 步，如图 3.9 (b) 所示。接下来两个指针以相同的速度在链表上向前移动，直到它们相遇。它们相遇的节点正好是环的入口节点，如图 3.9 (c) 所示。

剩下的问题是如何得到环中节点的数目。我们在前面提到判断一个链表里是否有环时用到了一快一慢两个指针。如果两个指针相遇，则表明链表中存在环。两个指针相遇的节点一定是在环中。可以从这个节点出发，一边继续向前移动一边计数，当再次回到这个节点时，就可以得到环中节点数了。

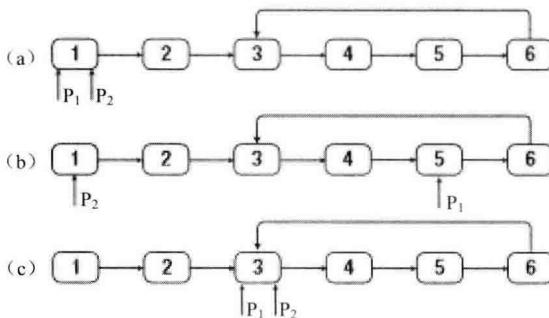


图 3.9 在有环的链表中找到环的入口节点的步骤

注：(1) 指针 P_1 和 P_2 在初始化时都指向链表的头节点。(2) 由于环中有 4 个节点，所以指针 P_1 先在链表上向前移动 4 步。(3) 指针 P_1 和 P_2 以相同的速度在链表上向前移动，直到它们相遇。它们相遇的节点就是环的入口节点。

下面代码中的函数 `MeetingNode` 在链表中存在环的前提下找到一快一慢两个指针相遇的节点。

```
ListNode* MeetingNode(ListNode* pHead)
{
    if(pHead == nullptr)
        return nullptr;

    ListNode* pSlow = pHead->m_pNext;
    if(pSlow == nullptr)
        return nullptr;

    ListNode* pFast = pSlow->m_pNext;
    while(pFast != nullptr && pSlow != nullptr)
    {
        if(pFast == pSlow)
            return pFast;

        pSlow = pSlow->m_pNext;

        pFast = pFast->m_pNext;
        if(pFast != nullptr)
            pFast = pFast->m_pNext;
    }

    return nullptr;
}
```

如果链表中不存在环，那么函数 `MeetingNode` 返回 `nullptr`。

在找到环中任意一个节点之后，就能得出环中的节点数目，并找到环的入口节点。相应的代码如下：

```
ListNode* EntryNodeOfLoop(ListNode* pHead)
{
    ListNode* meetingNode = MeetingNode(pHead);
    if(meetingNode == nullptr)
        return nullptr;

    // 得到环中节点的数目
    int nodesInLoop = 1;
    ListNode* pNode1 = meetingNode;
    while(pNode1->m_pNext != meetingNode)
    {
        pNode1 = pNode1->m_pNext;
        ++nodesInLoop;
    }

    // 先移动 pNode1，次数为环中节点的数目
    pNode1 = pHead;
    for(int i = 0; i < nodesInLoop; ++i)
        pNode1 = pNode1->m_pNext;

    // 再移动 pNode1 和 pNode2
    ListNode* pNode2 = pHead;
    while(pNode1 != pNode2)
    {
        pNode1 = pNode1->m_pNext;
        pNode2 = pNode2->m_pNext;
    }

    return pNode1;
}
```



源代码：

本题完整的源代码（含单元测试用例）：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/23_EntryNodeInListLoop



测试用例：

- 功能测试（链表中包含或者不包含环；链表中有多个或者只有一个节点）。
- 特殊输入测试（链表头节点为 `nullptr` 指针）。



本题考点：

- 考查应聘者对链表的理解。
- 考查应聘者所写代码的鲁棒性。鲁棒性是解决这道题的关键所在。如果应聘者写出的代码有着多处崩溃的潜在风险，那么他是很难通过这轮面试的。
- 考查应聘者分析问题的能力。把一个问题分解成几个简单的步骤，是一种常用的解决复杂问题的方法。为了解决这个问题，我们可以把它分解成3个步骤：找出环中任意一个节点；得到环中节点的数目；找到环的入口节点。更多关于分解让复杂问题简单化的讨论详见4.4节。

面试题 24：反转链表

题目：定义一个函数，输入一个链表的头节点，反转该链表并输出反转后链表的头节点。链表节点定义如下：

```
struct ListNode
{
    int         m_nKey;
    ListNode*  m_pNext;
};
```

解决与链表相关的问题总是有大量的指针操作，而指针操作的代码总是容易出错的。很多面试官喜欢出与链表相关的问题，就是想通过指针操作来考查应聘者的编码功底。为了避免出错，我们最好先进行全面的分析。在实际软件开发周期中，设计的时间通常不会比编码的时间短。在面试的时候我们不要急于动手写代码，而是一开始仔细分析和设计，这将会给面试官留下很好的印象。与其很快写出一段漏洞百出的代码，倒不如仔细分析再写出鲁棒的代码。

为了正确地反转一个链表，需要调整链表中指针的方向。为了将调整指针这个复杂的过程分析清楚，我们可以借助图形来直观地分析。在如图3.10(a)所示的链表中，*h*、*i*和*j*是3个相邻的节点。假设经过若干操作，我们已经把节点*h*之前的指针调整完毕，这些节点的*m_pNext*都指向前一个节点。接下来我们把*i*的*m_pNext*指向*h*，此时的链表结构如图3.10(b)所示。

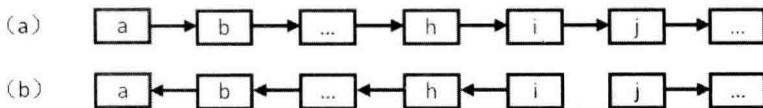


图 3.10 反转链表中节点的 `m_pNext` 指针导致链表出现断裂

注: (a) 一个链表。(b) 把 *i* 之前所有节点的 `m_pNext` 都指向前一个节点, 导致链表在节点 *i*、*j* 之间断裂。

不难注意到, 由于节点 *i* 的 `m_pNext` 指向了它的前一个节点, 导致我们无法在链表中遍历到节点 *j*。为了避免链表在节点 *i* 处断开, 我们需要在调整节点 *i* 的 `m_pNext` 之前, 把节点 *j* 保存下来。

也就是说, 我们在调整节点 *i* 的 `m_pNext` 指针时, 除了需要知道节点 *i* 本身, 还需要知道 *i* 的前一个节点 *h*, 因为我们需要把节点 *i* 的 `m_pNext` 指向节点 *h*。同时, 我们还需要事先保存 *i* 的一个节点 *j*, 以防止链表断开。因此, 相应地我们需要定义 3 个指针, 分别指向当前遍历到的节点、它的前一个节点及后一个节点。

最后我们试着找到反转后链表的头节点。不难分析出反转后链表的头节点是原始链表的尾节点。什么节点是尾节点? 自然是 `m_pNext` 为 `nullptr` 的节点。

有了前面的分析, 我们不难写出如下代码:

```

ListNode* ReverseList(ListNode* pHead)
{
    ListNode* pReversedHead = nullptr;
    ListNode* pNode = pHead;
    ListNode* pPrev = nullptr;
    while(pNode != nullptr)
    {
        ListNode* pNext = pNode->m_pNext;

        if(pNext == nullptr)
            pReversedHead = pNode;

        pNode->m_pNext = pPrev;
        pPrev = pNode;
        pNode = pNext;
    }

    return pReversedHead;
}
    
```

在面试过程中, 我们发现应聘者所写的代码中经常出现如下 3 种问题:

- 输入的链表头指针为 `nullptr` 或者整个链表只有一个节点时，程序立即崩溃。
- 反转后的链表出现断裂。
- 返回的反转之后的头节点不是原始链表的尾节点。

在实际面试的时候，不同应聘者的思路各不相同，因此写出的代码也不一样。那么，应聘者如何才能及时发现并纠正代码中的问题，以确保不犯上述错误呢？一个很好的办法就是提前想好测试用例。在写出代码之后，立即用事先准备好的测试用例检查测试。如果面试是以手写代码的方式，那也要在心里默默运行代码进行单元测试。只有确保代码通过测试之后，再提交给面试官。我们要记住一点：自己多花时间找出问题并修正问题，比在面试官找出问题之后再去慌慌张张修改代码要好得多。其实面试官检查应聘者所写代码的方法也是用他事先准备好的测试用例来进行测试。如果应聘者能够想到这些测试用例，并用它们来检查测试自己的代码，那就能够保证有备无患、万无一失了。

以这道题为例，我们至少应该想到以下几类测试用例对代码进行功能测试：

- 输入的链表头指针是 `nullptr`。
- 输入的链表只有一个节点。
- 输入的链表有多个节点。

如果我们确信代码能够通过这3类测试用例的测试，那么我们就有很大的把握能够通过这轮面试。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/24_ReverseList



测试用例：

- 功能测试（输入的链表含有多个节点；链表中只有一个节点）。
- 特殊输入测试（链表头节点为 `nullptr` 指针）。



本题考点：

- 考查应聘者对链表、指针的编程能力。
- 特别注重考查应聘者思维的全面性及写出来的代码的鲁棒性。



本题扩展：

用递归实现同样的反转链表的功能。

面试题 25：合并两个排序的链表

题目：输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。例如，输入图 3.11 中的链表 1 和链表 2，则合并之后的升序链表如链表 3 所示。链表节点定义如下：

```
struct ListNode
{
    int      m_nValue;
    ListNode* m_pNext;
};
```

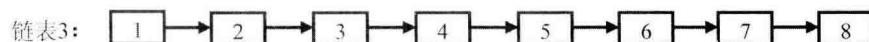
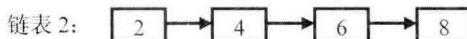
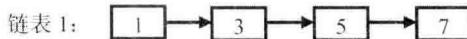


图 3.11 合并两个排序链表的过程

注：链表 1 和链表 2 是两个递增排序的链表，合并这两个链表得到的升序链表为链表 3。

这是一道经常被各公司采用的面试题。在面试过程中，我们发现应聘者最容易犯两种错误：一是在写代码之前没有想清楚合并的过程，最终合并出来的链表要么中间断开了、要么并没有做到递增排序；二是代码在鲁棒性方面存在问题，程序一旦有特殊的输入（如空链表）就会崩溃。接下来分析如何解决这两个问题。

首先分析合并两个链表的过程。我们的分析从合并两个链表的头节点开始。链表 1 的头节点的值小于链表 2 的头节点的值，因此链表 1 的头节

点将是合并后链表的头节点，如图3.12(a)所示。

我们继续合并两个链表中剩余的节点(图3.12中虚线框中的链表)。在两个链表中剩下的节点依然是排序的，因此合并这两个链表的步骤和前面的步骤是一样的。我们还是比较两个头节点的值。此时链表2的头节点的值小于链表1的头节点的值，因此链表2的头节点的值将是合并剩余节点得到的链表的头节点。我们把这个节点和前面合并链表时得到的链表的尾节点(值为1的节点)链接起来，如图3.12(b)所示。

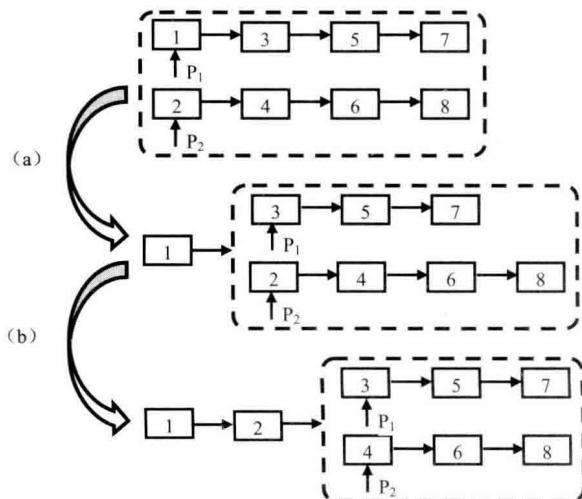


图3.12 合并两个递增链表的过程

注：(a) 链表1的头节点的值小于链表2的头节点的值，因此链表1的头节点是合并后链表的头节点。(b) 在剩余的节点中，链表2的头节点的值小于链表1的头节点的值，因此链表2的头节点是剩余节点的头节点，把这个节点和之前已经合并好的链表的尾节点链接起来。

当我们得到两个链表中值较小的头节点并把它链接到已经合并的链表之后，两个链表剩余的节点依然是排序的，因此合并的步骤和之前的步骤是一样的。这就是典型的递归过程，我们可以定义递归函数完成这一合并过程。

接下来我们来解决鲁棒性的问题。每当代码试图访问空指针指向的内存时程序就会崩溃，从而导致鲁棒性问题。在本题中一旦输入空的链表就会引入空的指针，因此我们要对空链表单独处理。当第一个链表是空链表，也就是它的头节点是一个空指针时，那么把它和第二个链表合并，显然合并的结果就是第二个链表。同样，当输入的第二个链表的头节点是空指针

的时候，我们把它和第一个链表合并得到的结果就是第一个链表。如果两个链表都是空链表，则合并的结果是得到一个空链表。

在我们想清楚合并的过程，并且知道哪些输入可能会引起鲁棒性问题之后，就可以动手写代码了。下面是一段参考代码：

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

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

    return pMergedHead;
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/25_MergeSortedLists



测试用例：

- 功能测试（输入的两个链表有多个节点；节点的值互不相同或者存在值相等的多个节点）。
- 特殊输入测试（两个链表的一个或者两个头节点为 `nullptr` 指针；两个链表中只有一个节点）。



本题考点：

- 考查应聘者分析问题的能力。解决这个问题需要大量的指针操作，应聘者如果没有透彻地分析问题形成清晰的思路，则很难写出正确的代码。
- 考查应聘者能不能写出鲁棒的代码。由于有大量指针操作，应聘者如果稍有不慎就会在代码中遗留很多与鲁棒性相关的隐患。建议应聘者在写代码之前全面分析哪些情况会引入空指针，并考虑清楚怎么处理这些空指针。

面试题 26：树的子结构

题目：输入两棵二叉树 A 和 B，判断 B 是不是 A 的子结构。二叉树节点的定义如下：

```
struct BinaryTreeNode
{
    double m_dbValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

例如图 3.13 中的两棵二叉树，由于 A 中有一部分子树的结构和 B 是一样的，因此 B 是 A 的子结构。

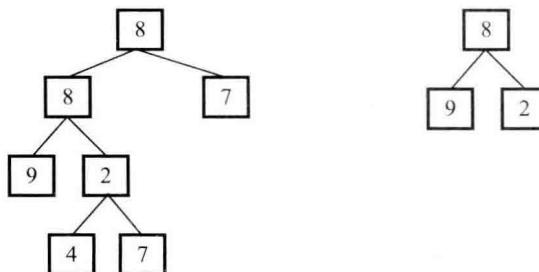


图 3.13 两棵二叉树 A 和 B，右边的树 B 是左边的树 A 的子结构

和链表相比，树中的指针操作更多也更复杂，因此与树相关的问题通常会比链表的要难。如果想加大面试的难度，则树的题目是很多面试官的选择。面对大量的指针操作，我们要更加小心，否则一不留神就会在代码中留下隐患。

现在回到这道题目本身。要查找树 A 中是否存在和树 B 结构一样的子

树，我们可以分成两步：第一步，在树 A 中找到和树 B 的根节点的值一样的节点 R；第二步，判断树 A 中以 R 为根节点的子树是不是包含和树 B 一样的结构。

以上面的两棵树为例来详细分析这个过程。首先试着在树 A 中找到值为 8（树 B 的根节点的值）的节点。从树 A 的根节点开始遍历，我们发现它的根节点的值就是 8。接着判断树 A 的根节点下面的子树是不是含有和树 B 一样的结构，如图 3.14 所示。在树 A 中，根节点的左子节点的值是 8，而树 B 的根节点的左子节点是 9，对应的两个节点不同。

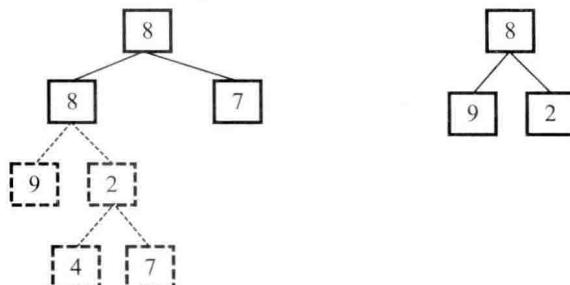


图 3.14 树 A 的根节点和树 B 的根节点的值相同，但树 A 的根节点下面（实线部分）的结构和树 B 的结构不一致

因此我们仍然需要遍历树 A，接着查找值为 8 的节点。我们在树的第二层中找到了一个值为 8 的节点，然后进行第二步判断，即判断这个节点下面的子树是否含有和树 B 一样的结构，如图 3.15 所示。于是我们遍历这个节点下面的子树，先后得到两个子节点 9 和 2，这和树 B 的结构完全相同。此时我们在树 A 中找到了一棵和树 B 的结构一样的子树，因此树 B 是树 A 的子结构。



图 3.15 在树 A 中找到第二个值为 8 的节点，该节点下面（实线部分）的结构和树 B 的结构一致

第一步在树 A 中查找与根节点的值一样的节点，这实际上就是树的遍历。对二叉树这种数据结构熟悉的读者自然知道可以用递归的方法去遍历，也可以用循环的方法去遍历。由于递归的代码实现比较简洁，面试时如果没有特别要求，那么我们通常会采用递归的方式。下面是参考代码：

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

    if(pRoot1 != nullptr && pRoot2 != nullptr)
    {
        if(Equal(pRoot1->m_dbValue, pRoot2->m_dbValue))
            result = DoesTree1HaveTree2(pRoot1, pRoot2);
        if(!result)
            result = HasSubtree(pRoot1->m_pLeft, pRoot2);
        if(!result)
            result = HasSubtree(pRoot1->m_pRight, pRoot2);
    }

    return result;
}
```

在面试的时候，我们一定要注意边界条件的检查，即检查空指针。当树 A 或树 B 为空的时候，定义相应的输出。如果没有检查并进行相应的处理，则程序非常容易崩溃，这是面试时非常忌讳的事情。

在上述代码中，我们递归调用 HasSubtree 遍历二叉树 A。如果发现某一节点的值和树 B 的头节点的值相同，则调用 DoesTree1HaveTree2，进行第二步判断。

第二步是判断树 A 中以 R 为根节点的子树是不是和树 B 具有相同的结构。同样，我们也可以用递归的思路来考虑：如果节点 R 的值和树 B 的根节点不相同，则以 R 为根节点的子树和树 B 肯定不具有相同的节点；如果它们的值相同，则递归地判断它们各自的左右节点的值是不是相同。递归的终止条件是我们到达了树 A 或者树 B 的叶节点。参考代码如下：

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

    if(pRoot1 == nullptr)
        return false;

    if(!Equal(pRoot1->m_dbValue, pRoot2->m_dbValue))
        return false;

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

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

我们注意到上述代码有多处判断一个指针是不是 `nullptr`，这样做是为了避免试图访问空指针而造成程序崩溃，同时也设置了递归调用的退出条件。在写遍历树的代码的时候一定要高度警惕，在每一处需要访问地址的时候都要问自己这个地址有没有可能是 `nullptr`、如果是 `nullptr` 则该怎么处理。



面试小提示：

与二叉树相关的代码有大量的指针操作，在每次使用指针的时候，我们都要问自己这个指针有没有可能是 `nullptr`，如果是 `nullptr` 则该怎么处理。

为了确保自己所写的代码完整、正确，在写出代码之后，应聘者至少要用几个测试用例来检验自己的程序：树 A 和树 B 的头节点有一个或者两个都是空指针；在树 A 和树 B 中所有节点都只有左子节点或者右子节点；树 A 和树 B 的节点中含有分叉。只有这样才能写出让面试官满意的鲁棒代码。

一个细节值得我们注意：本题中节点中值的类型为 `double`。在判断两个节点的值是不是相等时，不能直接写 `pRoot1->m_dbValue == pRoot2->m_dbValue`，这是因为在计算机内表示小数时（包括 `float` 和 `double` 型小数）都有误差。判断两个小数是否相等，只能判断它们之差的绝对值是不是在一个很小的范围内。如果两个数相差很小，就可以认为它们相等。这就是我们定义函数 `Equal` 的原因。

```
bool Equal(double num1, double num2)
{
    if((num1 - num2 > -0.0000001) && (num1 - num2 < 0.0000001))
        return true;
    else
        return false;
}
```



面试小提示：

由于计算机表示小数（包括 `float` 和 `double` 型小数）都有误差，我们不能直接用等号（`==`）判断两个小数是否相等。如果两个小数的差的绝对值很小，如小于 0.0000001，就可以认为它们相等。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/26_SubstructureInTree



测试用例：

- 功能测试（树 A 和树 B 都是普通的二叉树；树 B 是或者不是树 A 的子结构）。
- 特殊输入测试（两棵二叉树的一个或者两个根节点为 `nullptr` 指针；二叉树的所有节点都没有左子树或者右子树）。



本题考点：

- 考查应聘者对二叉树遍历算法的理解及递归编程能力。
- 考查应聘者所写代码的鲁棒性。本题的代码中含有大量的指针操作，稍有不慎程序就会崩溃。应聘者需要采用防御性编程的方式，每次访问指针地址之前都要考虑这个指针有没有可能是 `nullptr`。

3.5 本章小结

本章从规范性、完整性和鲁棒性 3 个方面介绍了如何在面试时写出高质量的代码，如图 3.16 所示。

大多数面试都要求应聘者在白纸或者白板上写代码。应聘者在编码的时候要注意规范性，尽量清晰地书写每个字母，通过缩进和对齐括号让代码布局合理，同时合理命名代码中的变量和函数。

最好在编码之前全面考虑所有可能的输入，确保写出的代码在完成了基本功能之外，还考虑了边界条件，并做好了错误处理。只有全面考虑到这 3 个方面的代码才是完整的代码。

另外，要确保自己写出的程序不会轻易崩溃。平时在写代码的时候，

应聘者最好养成防御性编程的习惯，在函数入口判断输入是否有效，并对各种无效输入做好相应的处理。

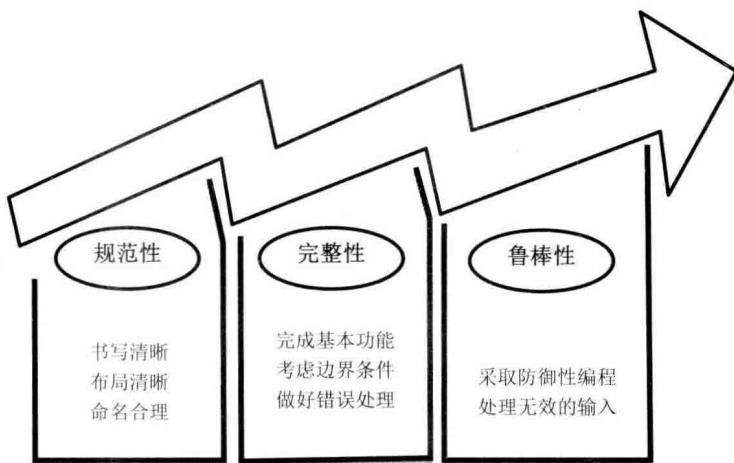


图 3.16 从规范性、完整性和鲁棒性 3 个方面提高代码的质量

第4章

解决面试题的思路

4.1

面试官谈面试思路

“编码前讲自己的思路是一个考查指标。一个合格的应聘者应该在他做事之前明白自己要做的事情究竟是什么，以及该怎么做。一开始就编码的人员，除非后面表现得非常优秀，否则很容易通不过。”

——殷焰（支付宝，高级安全测试工程师）

“让应聘者给我讲具体的问题分析过程，经常会要求他证明。”

——张晓禹（百度，技术经理）

“我个人比较倾向于让应聘者在写代码之前解释他的思路。应聘者如果没有想清楚就动手，则本身就不是太好。应聘者可以采用举例子、画图等多种方式，解释清楚问题本身和问题解决方案是关键。”

——何幸杰（SAP，高级工程师）

“对于比较复杂的算法和设计，一般来讲最好在开始写代码之前讲清楚思路和设计。”

——尧敏（淘宝，资深经理）

“喜欢应聘者先讲清思路。如果觉察到方案的错误和漏洞，那么我会让他证明是否正确，主要是希望他能在分析的过程中发现这些错误和漏洞并加以改正。”

——陈黎明（微软，SDE II）

“喜欢应聘者在写代码之前先讲思路，举例子和画图都是很好的方法。”

——田超（微软，SDE II）

4.2 画图让抽象问题形象化

画图是在面试过程中应聘者用来帮助自己分析、推理的常用手段。很多面试题很抽象，不容易找到解决办法。这时不妨画出一些与题目相关的图形，借以辅助自己观察和思考。图形能使抽象的问题具体化、形象化，应聘者说不定通过几幅图形就能找到规律，从而找到问题的解决方案。

有不少与数据结构相关的问题，比如二叉树、二维数组、链表等问题，都可以采用画图的方法来分析。很多时候空想未必能想明白题目中隐含的规律和特点，随手画几幅图却能让我们轻易地找到窍门。比如在面试题27“二叉树的镜像”中，我们画几幅二叉树的图就能发现，求树的镜像的过程其实就是在遍历树的同时交换非叶节点的左、右子节点。在面试题29“顺时针打印矩阵”中，我们画图之后很容易就发现可以把矩阵分解成若干个圆圈，然后从外向内打印每个圆圈。面试的时候很多人都会在边界条件上犯错误（因为最后一圈可能退化而不是一个完整的圈），如果画几幅示意图，就能够很容易找到最后一圈退化的规律。对于面试题35“复杂链表的复制”，如果能够画出每一步操作时的指针操作，那么接下来写代码就会容易得多。

在面试的时候，应聘者需要向面试官解释自己的思路。对于复杂的问题，应聘者光用语言未必能够说得清楚。这时候可以画出几幅图形，一边看着图形一边讲解，面试官就能更加轻松地理解应聘者的思路。这对应聘者是有益的，因为面试官会觉得他具有很好的沟通交流能力。

面试题 27：二叉树的镜像

题目：请完成一个函数，输入一棵二叉树，该函数输出它的镜像。**二叉树节点的定义如下：**

```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

树的镜像对很多人来说是一个新的概念，我们未必能够一下子想出求树的镜像的方法。为了能够形成直观的印象，我们可以自己画一棵二叉树，然后根据照镜子的经验画出它的镜像。如图 4.1 中右边的二叉树就是左边的树的镜像。

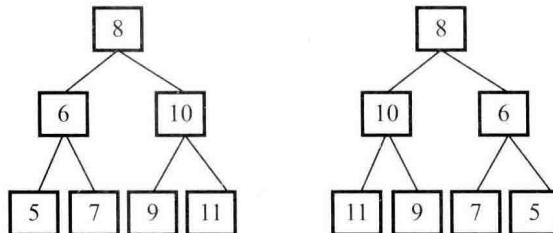


图 4.1 两棵互为镜像的二叉树

仔细分析这两棵树的特点，看看能不能总结出求镜像的步骤。这两棵树的根节点相同，但它们的左、右两个子节点交换了位置。因此，我们不妨先在树中交换根节点的两个子节点，就得到图 4.2 中的第二棵树。

交换根节点的两个子节点之后，我们注意到值为 10、6 的节点的子节点仍然保持不变，因此我们还需要交换这两个节点的左、右子节点。交换之后的结果分别是图 4.2 中的第三棵树和第四棵树。做完这两次交换之后，我们已经遍历完所有的非叶节点。此时变换之后的树刚好就是原始树的镜像。

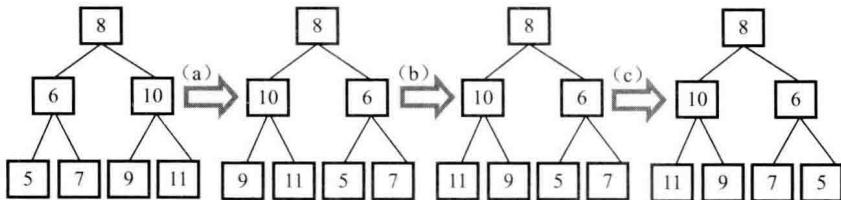


图 4.2 求二叉树镜像的过程

注：(a) 交换根节点的左、右子树；(b) 交换值为 10 的节点的左、右子节点；(c) 交换值为 6 的节点的左、右子节点。

总结上面的过程，我们得出求一棵树的镜像的过程：先前序遍历这棵树的每个节点，如果遍历到的节点有子节点，就交换它的两个子节点。当交换完所有非叶节点的左、右子节点之后，就得到了树的镜像。

想清楚了这种思路，我们就可以动手写代码了。参考代码如下：

```

void MirrorRecursively (BinaryTreeNode *pNode)
{
    if(pNode == nullptr)
        return;
    if(pNode->m_pLeft == nullptr && pNode->m_pRight == nullptr)
        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);
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/27_MirrorOfBinaryTree



测试用例：

- 功能测试（普通的二叉树；二叉树的所有节点都没有左子树或者右子树；只有一个节点的二叉树）。
- 特殊输入测试（二叉树的根节点为 nullptr 指针）。



本题考点：

- 考查应聘者对二叉树的理解。本题实质上是利用树的遍历算法解决问题。
- 考查应聘者的思维能力。树的镜像是一个抽象的概念，应聘者需要在短时间内想清楚求镜像的步骤并转换为代码。应聘者可以通过画图把抽象的问题形象化，这有助于其快速找到解题思路。



本题扩展：

上面的代码是用递归实现的。如果要求用循环，则该如何实现？

面试题 28：对称的二叉树

题目：请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。例如，在如图 4.3 所示的 3 棵二叉树中，第一棵二叉树是对称的，而另外两棵不是。

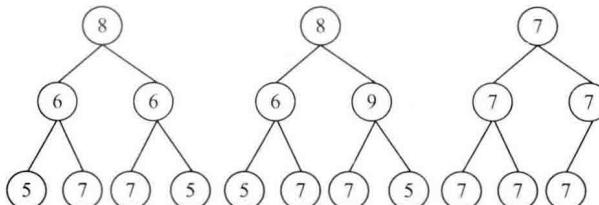


图 4.3 3 棵二叉树，其中第一棵二叉树是对称的，另外两棵不是

通常我们有 3 种不同的二叉树遍历算法，即前序遍历、中序遍历和后序遍历。在这 3 种遍历算法中，都是先遍历左子节点再遍历右子节点。我们是否可以定义一种遍历算法，先遍历右子节点再遍历左子节点？比如我们针对前序遍历定义一种对称的遍历算法，即先遍历父节点，再遍历它的右子节点，最后遍历它的左子节点。

如果用前序遍历算法遍历图 4.3 中的第一棵二叉树，则遍历序列是 {8, 6, 5, 7, 6, 7, 5}。如果用我们定义的针对前序遍历的对称遍历算法，则得到的遍历序列是 {8, 6, 5, 7, 6, 7, 5}。我们注意到这两个序列是一样的。

图 4.3 中第二棵二叉树的前序遍历序列为 {8, 6, 5, 7, 9, 7, 5}，而相应的

对称前序遍历序列为{8, 9, 5, 7, 6, 7, 5}。在这两个序列中，第二步和第五步是不一样的。

第三棵二叉树有些特殊，它所有节点的值都是一样的。它的前序遍历序列是{7, 7, 7, 7, 7, 7}，前序遍历的对称遍历序列也是{7, 7, 7, 7, 7, 7}。这两个序列是一样的，可显然第三棵二叉树不是对称的。怎样才能正确地判断这种类型的二叉树呢？只要我们在遍历二叉树时把遇到的 `nullptr` 指针也考虑进来就行了。

比如第三棵二叉树的前序遍历序列在考虑 `nullptr` 指针之后为{7, 7, 7, `nullptr`, `nullptr`, 7, `nullptr`, `nullptr`, 7, 7, `nullptr`, `nullptr`, `nullptr`}。序列的前面3个7对应的是从根节点开始沿着指向左子节点的指针遍历经过的3个节点，接下来两个 `nullptr` 指针对应的是第三层第一个节点的两个子节点，其他的节点请读者自己分析。前序遍历的对称遍历序列为{7, 7, `nullptr`, 7, `nullptr`, `nullptr`, 7, 7, `nullptr`, `nullptr`, 7, `nullptr`, `nullptr`}。这两个序列从第三步开始就不一致了。

我们发现可以通过比较二叉树的前序遍历序列和对称前序遍历序列来判断二叉树是不是对称的。如果两个序列是一样的，那么二叉树就是对称的。

```
bool isSymmetrical(TreeNode* pRoot)
{
    return isSymmetrical(pRoot, pRoot);
}

bool isSymmetrical(TreeNode* pRoot1, TreeNode* pRoot2)
{
    if(pRoot1 == nullptr && pRoot2 == nullptr)
        return true;

    if(pRoot1 == nullptr || pRoot2 == nullptr)
        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);
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/28_SymmetricalBinaryTree



测试用例：

- 功能测试（对称的二叉树；因结构而不对称的二叉树；结构对称但节点的值不对称的二叉树）。
- 特殊输入测试（二叉树的根节点为 `nullptr` 指针；只有一个节点的二叉树；所有节点的值都相同的二叉树）。



本题考点：

- 考查应聘者对二叉树的理解。本题实质上是利用树的遍历算法解决问题。
- 考查应聘者的思维能力。树的对称是一个抽象的概念，应聘者需要在短时间内想清楚判断对称的步骤并转换为代码。应聘者可以通过画图把抽象的问题形象化，这有助于其快速找到解题思路。

面试题 29：顺时针打印矩阵

题目：输入一个矩阵，按照从外向里以顺时针的顺序依次打印出每一个数字。例如，如果输入如下矩阵：

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

则依次打印出数字 1,2,3,4,8,12,16,15,14,13,9,5,6,7,11,10。

这道题完全没有涉及复杂的数据结构或者高级的算法，看起来是一个很简单的问题。但实际上解决这个问题时会在代码中包含多个循环，并且需要判断多个边界条件。如果在把问题考虑得很清楚之前就开始写代码，则不可避免地会越写越混乱。因此解决这个问题的关键在于先形成清晰的思路，并把复杂的问题分解成若干个简单的问题。

当我们遇到一个复杂问题的时候，可以用图形来帮助思考。由于是以从外圈到内圈的顺序依次打印的，所以我们可以把矩阵想象成若干个圈，如图 4.4 所示。我们可以用一个循环来打印矩阵，每次打印矩阵中的一个圈。

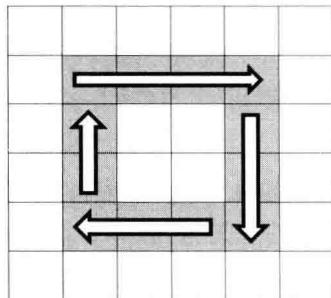


图 4.4 把矩阵看成由若干个顺时针方向的圈组成

接下来分析循环结束的条件。假设这个矩阵的行数是 `rows`，列数是 `columns`。打印第一圈的左上角的坐标是 $(0, 0)$ ，第二圈的左上角的坐标是 $(1, 1)$ ，以此类推。我们注意到，左上角的坐标中行标和列标总是相同的，于是可以在矩阵中选取左上角为 $(start, start)$ 的一圈作为我们分析的目标。

对于一个 5×5 的矩阵而言，最后一圈只有一个数字，对应的坐标为 $(2, 2)$ 。我们发现 $5 > 2 \times 2$ 。对于一个 6×6 的矩阵而言，最后一圈有 4 个数字，其左上角的坐标仍然为 $(2, 2)$ 。我们发现 $6 > 2 \times 2$ 依然成立。于是可以得出，让循环继续的条件是 $columns > startX \times 2$ 并且 $rows > startY \times 2$ 。所以我们可以用如下的循环来打印矩阵：

```
void PrintMatrixClockwisely(int** numbers, int columns, int rows)
{
    if(numbers == nullptr || columns <= 0 || rows <= 0)
        return;

    int start = 0;

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

        ++start;
    }
}
```

接着我们考虑如何打印一圈的功能，即如何实现 `PrintMatrixInCircle`。如图 4.4 所示，我们可以把打印一圈分为四步：第一步，从左到右打印一行；第二步，从上到下打印一列；第三步，从右到左打印一行；第四步，从下

到上打印一列。每一步我们根据起始坐标和终止坐标用一个循环就能打印出一行或者一列。

不过值得注意的是，最后一圈有可能退化成只有一行、只有一列，甚至只有一个数字，因此打印这样的一圈就不再需要四步。图 4.5 是几个退化的例子，打印一圈分别只需要三步、两步甚至一步。

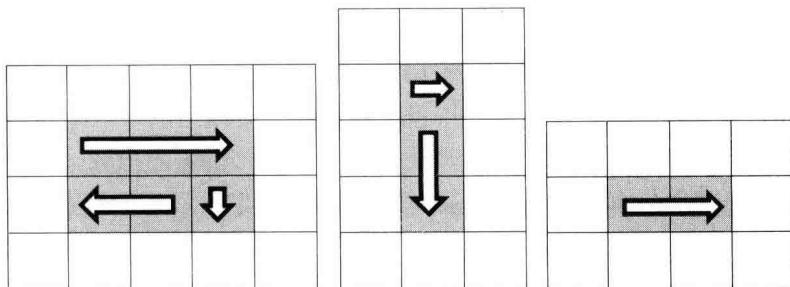


图 4.5 打印矩阵最里面一圈可能只需要三步、两步甚至一步

因此，我们要仔细分析打印时每一步的前提条件。第一步总是需要的，因为打印一圈至少有一步。如果只有一行，那就不用第二步了。也就是需要第二步的前提条件是终止行号大于起始行号。需要第三步打印的前提条件是圈内至少有两行两列，也就是说，除了要求终止行号大于起始行号，还要求终止列号大于起始列号。同理，需要打印第四步的前提条件是至少有三行两列，因此要求终止行号比起始行号至少大 2，同时终止列号大于起始列号。

通过上述分析，我们就可以写出如下代码：

```
void PrintMatrixInCircle(int** numbers, int columns, int rows, int start)
{
    int endX = columns - 1 - start;
    int endY = rows - 1 - start;

    // 从左到右打印一行
    for(int i = start; i <= endX; ++i)
    {
        int number = numbers[start][i];
        printNumber(number);
    }

    // 从上到下打印一列
    if(start < endY)
    {
        for(int i = start + 1; i <= endY; ++i)
        {
            int number = numbers[i][endX];
            printNumber(number);
        }
    }
}
```

```

        int number = numbers[i][endX];
        printNumber(number);
    }
}

// 从右到左打印一行
if(start < endX && start < endY)
{
    for(int i = endX - 1; i >= start; --i)
    {
        int number = numbers[endY][i];
        printNumber(number);
    }
}

// 从下到上打印一列
if(start < endX && start < endY - 1)
{
    for(int i = endY - 1; i >= start + 1; --i)
    {
        int number = numbers[i][start];
        printNumber(number);
    }
}
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/29_PrintMatrix



测试用例：

数组中有多行多列；数组中只有一行；数组中只有一列；数组中只有一行一列。



本题考点：

本题主要考查应聘者的思维能力。从外到内顺时针打印矩阵这个过程非常复杂，应聘者如何能很快地找出其规律并写出完整的代码，是解这道题的关键。当问题比较抽象不容易理解时，可以试着画几幅图形帮助理解，这样往往能更快地找到思路。

4.3

举例让抽象问题具体化

和上一节画图的方法一样，我们也可以借助举例模拟的方法来思考分析复杂的问题。当一眼看不出问题中隐藏的规律的时候，我们可以试着用一两个具体的例子模拟操作的过程，这样说不定就能通过具体的例子找到抽象的规律。比如面试题 31 “栈的压入、弹出序列”，很多人不能立即找到栈的压入和弹出规律。这时我们可以仔细分析一两个序列，一步一步模拟压入、弹出的操作，并从中总结出隐含的规律。面试题 33 “二叉搜索树的后序遍历序列”也类似，我们同样可以通过一两个具体的序列找到后序遍历的规律。

具体的例子也可以帮助我们向面试官解释算法思路。算法通常是很抽象的，用语言不容易表述得很清楚，我们可以考虑举一两个具体的例子，告诉面试官我们的算法是怎么一步步处理这个例子的。例如，在面试题 30 “包含 min 函数的栈”中，我们可以举例模拟压栈和弹出几个数字，分析每次操作之后数据栈、辅助栈和最小值各是什么。这样解释之后，面试官就能很清晰地理解我们的思路，同时他也会觉得我们有很好的沟通能力，能把复杂的问题用很简单的方式说清楚。

具体的例子还能帮助我们确保代码的质量。在面试中写完代码之后，应该先检查一遍，确保没有问题再交给面试官。怎么检查呢？我们可以运行几个测试用例。在分析问题的时候采用的例子就是测试用例。我们可以把这些例子当作测试用例，在心里模拟运行，看每一步操作之后的结果和我们预期的是不是一致。如果每一步的结果都和事先预计的一致，那我们就能确保代码的正确性了。

面试题 30：包含 min 函数的栈

题目：定义栈的数据结构，请在该类型中实现一个能够得到栈的最小元素的 min 函数。在该栈中，调用 min、push 及 pop 的时间复杂度都是 $O(1)$ 。

看到这个问题，我们的第一反应可能是每次压入一个新元素进栈时，将栈里的所有元素排序，让最小的元素位于栈顶，这样就能在 $O(1)$ 时间内得到最小元素了。但这种思路不能保证最后压入栈的元素能够最先出栈，因此这

个数据结构已经不是栈了。

我们接着想到在栈里添加一个成员变量存放最小的元素。每次压入一个新元素进栈的时候，如果该元素比当前最小的元素还要小，则更新最小元素。面试官听到这种思路之后就会问：如果当前最小的元素被弹出栈了，那么如何得到下一个最小的元素呢？

分析到这里我们发现，仅仅添加一个成员变量存放最小元素是不够的，也就是说当最小元素被弹出栈的时候，我们希望能够得到次小元素。因此，在压入这个最小元素之前，我们要把次小元素保存起来。

是不是可以把每次的最小元素（之前的最小元素和新压入栈的元素两者的较小值）都保存起来放到另外一个辅助栈里呢？我们不妨举几个例子来分析一下把元素压入或者弹出栈的过程，如表4.1所示。

首先往空的数据栈里压入数字3，显然现在3是最小值，我们也把这个最小值压入辅助栈。接下来往数据栈里压入数字4。由于4大于之前的最小值，因此我们仍然往辅助栈里压入数字3。第三步继续往数据栈里压入数字2。由于2小于之前的最小值3，因此我们把最小值更新为2，并把2压入辅助栈。同样，当压入数字1时，也要更新最小值，并把新的最小值1压入辅助栈。

表4.1 栈内压入3,4,2,1之后接连两次弹出栈顶数字再压入0时，数据栈、辅助栈和最小值的状态

步 骤	操 作	数 据 栈	辅 助 栈	最 小 值
1	压入 3	3	3	3
2	压入 4	3, 4	3, 3	3
3	压入 2	3, 4, 2	3, 3, 2	2
4	压入 1	3, 4, 2, 1	3, 3, 2, 1	1
5	弹出	3, 4, 2	3, 3, 2	2
6	弹出	3, 4	3, 3	3
7	压入 0	3, 4, 0	3, 3, 0	0

从表4.1中可以看出，如果每次都把最小元素压入辅助栈，那么就能保证辅助栈的栈顶一直都是最小元素。当最小元素从数据栈内被弹出之后，同时弹出辅助栈的栈顶元素，此时辅助栈的新栈顶元素就是下一个最小值。比如在第四步之后，栈内的最小元素是1。当第五步在数据栈内弹出1后，

我们把辅助栈的栈顶弹出，辅助栈的栈顶元素 2 就是新的最小元素。接下来继续弹出数据栈和辅助栈的栈顶之后，数据栈还剩下 3、4 两个数字，3 是最小值。此时位于辅助栈的栈顶数字正好也是 3，的确是最小值。这说明我们的思路是正确的。

当我们想清楚上述过程之后，就可以写代码了。下面是 3 个关键函数 push、pop 和 min 的参考代码。在代码中，`m_data` 是数据栈，而 `m_min` 是辅助栈。

```
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();
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/30_MinInStack



测试用例：

- 新压入栈的数字比之前的最小值大。

- 新压入栈的数字比之前的最小值小。
- 弹出栈的数字不是最小元素。
- 弹出栈的数字是最小元素。



本题考点：

- 考查应聘者分析复杂问题的思维能力。在面试的时候，很多应聘者都止步于添加一个变量保存最小元素的思路。其实只要举个例子，多做几次入栈、出栈的操作就能看出问题，并想到也要把最小元素用另外的辅助栈保存。当我们面对一个抽象复杂的问题的时候，可以用几个具体的例子来找出规律。找到规律之后再解决问题，就容易多了。
- 考查应聘者对栈的理解。

面试题 31：栈的压入、弹出序列

题目：输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如，序列{1,2,3,4,5}是某栈的压栈序列，序列{4,5,3,2,1}是该压栈序列对应的一个弹出序列，但{4,3,5,1,2}就不可能是该压栈序列的弹出序列。

解决这个问题很直观的想法就是建立一个辅助栈，把输入的第一个序列中的数字依次压入该辅助栈，并按照第二个序列的顺序依次从该栈中弹出数字。

以弹出序列{4,5,3,2,1}为例分析压栈和弹出的过程。第一个希望被弹出的数字是4，因此4需要先压入辅助栈。压入栈的顺序由压栈序列确定，也就是在把4压入栈之前，数字1,2,3都需要先压入栈。此时栈里包含4个数字，分别是1,2,3,4，其中4位于栈顶。把4弹出栈后，剩下的3个数字是1、2和3。接下来希望被弹出的数字是5，由于它不是栈顶数字，因此我们接着在第一个序列中把4以后的数字压入辅助栈，直到压入数字5。这时候5位于栈顶，就可以被弹出来了。接下来希望被弹出的3个数字依次是3、2和1。由于每次操作前它们都位于栈顶，因此直接弹出即可。表4.2总结了本例中入栈和出栈的步骤。

表 4.2 压栈序列为{1,2,3,4,5}、弹出序列为{4,5,3,2,1}对应的压栈和弹出过程

步 骤	操 作	栈	弹出数字	步 骤	操 作	栈	弹出数字
1	压入 1	1		6	压入 5	1, 2, 3, 5	
2	压入 2	1, 2		7	弹出	1, 2, 3	5
3	压入 3	1, 2, 3		8	弹出	1, 2	3
4	压入 4	1, 2, 3, 4		9	弹出	1	2
5	弹出	1, 2, 3	4	10	弹出		1

接下来再分析弹出序列{4,3,5,1,2}。第一个弹出的数字4的情况和前面一样。把4弹出之后，3位于栈顶，可以直接弹出。接下来希望弹出的数字是5，由于5不是栈顶数字，到压栈序列里把没有压栈的数字压入辅助栈，直至遇到数字5。把数字5压入栈之后，5就位于栈顶了，可以弹出。此时栈内有两个数字1和2，其中2位于栈顶。由于接下来需要弹出的数字是1，但1不在栈顶，我们需要从压栈序列中尚未压入栈的数字中去搜索这个数字。但此时压栈序列中的所有数字都已经压入栈了，所以该序列不是序列{1,2,3,4,5}对应的弹出序列。表4.3总结了这个例子中压栈和弹出的过程。

表 4.3 一个压入顺序为{1,2,3,4,5}的栈没有一个弹出序列为{4,3,5,1,2}

步 骤	操 作	栈	弹出数字	步 骤	操 作	栈	弹出数字
1	压入 1	1		6	弹出	1, 2	3
2	压入 2	1, 2		7	压入 5	1, 2, 5	
3	压入 3	1, 2, 3		8	弹出	1, 2	5
4	压入 4	1, 2, 3, 4		下一个弹出的是1，但1不在栈顶，压栈序列的数字都已入栈。操作无法继续			
5	弹出	1, 2, 3	4				

总结上述入栈、出栈的过程，我们可以找到判断一个序列是不是栈的弹出序列的规律：如果下一个弹出的数字刚好是栈顶数字，那么直接弹出；如果下一个弹出的数字不在栈顶，则把压栈序列中还没有入栈的数字压入辅助栈，直到把下一个需要弹出的数字压入栈顶为止；如果所有数字都压入栈后仍然没有找到下一个弹出的数字，那么该序列不可能是一个弹出序列。

形成了清晰的思路之后，我们就可以动手写代码了。下面是一段参考代码：

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

```

if(pPush != nullptr && pPop != nullptr && nLength > 0)
{
    const int* pNextPush = pPush;
    const int* pNextPop = pPop;

    std::stack<int> stackData;

    while(pNextPop - pPop < nLength)
    {
        while(stackData.empty() || stackData.top() != *pNextPop)
        {
            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;
}

```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/31_StackPushPopOrder



测试用例：

- 功能测试（输入的两个数组含有多个数字或者只有一个数字；第二个数组是或者不是第一个数组表示的压入序列对应的栈的弹出序列）。
- 特殊输入测试（输入两个 nullptr 指针）。



本题考点：

- 考查应聘者分析复杂问题的能力。刚听到这道面试题的时候，很多人可能都没有思路。这时候可以通过举一两个例子，一步步分析压栈、弹出的过程，从中找出规律。
- 考查应聘者对栈的理解。

面试题 32：从上到下打印二叉树

题目一：分行从上到下打印二叉树

从上到下打印出二叉树的每个节点，同一层的节点按照从左到右的顺序打印。例如，输入图 4.6 中的二叉树，则依次打印出 8,6,10,5,7,9,11。二叉树节点的定义如下：

```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

这道题实质是考查树的遍历算法，只是这种遍历不是我们熟悉的前序、中序或者后序遍历。由于我们不太熟悉这种按层遍历的方法，可能一下子也想不清楚遍历的过程。那面试的时候怎么办呢？我们不妨先分析一下打印图 4.6 中的二叉树的过程。

因为按层打印的顺序决定应该先打印根节点，所以我们从树的根节点开始分析。为了接下来能够打印值为 8 的节点的两个子节点，我们应该在遍历到该节点时把值为 6 和 10 的两个节点保存到一个容器里，现在容器内就有两个节点了。按照从左到右打印的要求，我们先取出值为 6 的节点。打印出值 6 之后，把它的值分别为 5 和 7 的两个节点放入数据容器。此时数据容器中有 3 个节点，值分别为 10、5 和 7。接下来我们从数据容器中取出值为 10 的节点。注意到值为 10 的节点比值为 5 和 7 的节点先放入容器，此时又比这两个节点先取出，这就是我们通常所说的先入先出，因此不难看出这个数据容器应该是一个队列。由于值为 5、7、9、11 的节点都没有子节点，因此只要依次打印即可。整个打印过程如表 4.4 所示。

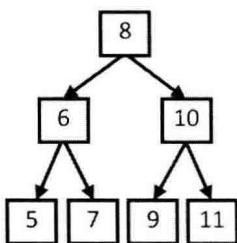


图 4.6 一棵二叉树，从上到下按层打印的顺序为 8,6,10,5,7,9,11

表 4.4 按层打印图 4.6 中的二叉树的过程

步 骤	操 作	队 列
1	打印节点 8	节点 6、节点 10
2	打印节点 6	节点 10、节点 5、节点 7
3	打印节点 10	节点 5、节点 7、节点 9、节点 11
4	打印节点 5	节点 7、节点 9、节点 11
5	打印节点 7	节点 9、节点 11
6	打印节点 9	节点 11
7	打印节点 11	

通过上面具体例子的分析，我们可以找到从上到下打印二叉树的规律：每次打印一个节点的时候，如果该节点有子节点，则把该节点的子节点放到一个队列的末尾。接下来到队列的头部取出最早进入队列的节点，重复前面的打印操作，直至队列中所有的节点都被打印出来。

既然我们已经确定数据容器是一个队列了，现在的问题就是如何实现队列。实际上我们无须自己动手实现，因为 STL 已经为我们实现了一个很好的 deque（两端都可以进出的队列）。下面是用 deque 实现的参考代码：

```

void PrintFromTopToBottom(BinaryTreeNode* pTreeRoot)
{
    if(!pTreeRoot)
        return;

    std::deque<BinaryTreeNode *> dequeTreeNode;
    dequeTreeNode.push_back(pTreeRoot);

    while(dequeTreeNode.size())
    {
        BinaryTreeNode *pNode = dequeTreeNode.front();
        dequeTreeNode.pop_front();
        // 打印 pNode
        // 将 pNode 的左右子节点依次加入队列
    }
}
  
```

```

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

if(pNode->m_pLeft)
    dequeTreeNode.push_back(pNode->m_pLeft);

if(pNode->m_pRight)
    dequeTreeNode.push_back(pNode->m_pRight);
}

}

```



源代码:

本题完整的源代码:

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/32_01_PrintTreeFromTopToBottom



测试用例:

- 功能测试（完全二叉树；所有节点只有左子树的二叉树；所有节点只有右子树的二叉树）。
- 特殊输入测试（二叉树根节点为 `nullptr` 指针；只有一个节点的二叉树）。



本题考点:

- 考查应聘者的思维能力。按层从上到下遍历二叉树，这对很多应聘者来说是一个新概念，要在短时间内想明白遍历的过程不是一件容易的事情。应聘者通过具体的例子找出其中的规律并想到基于队列的算法，是解决这个问题的关键所在。
- 考查应聘者对二叉树及队列的理解。



本题扩展:

如何广度优先遍历一幅有向图？同样也可以基于队列实现。树是图的一种特殊退化形式，从上到下按层遍历二叉树，从本质上来说就是广度优先遍历二叉树。

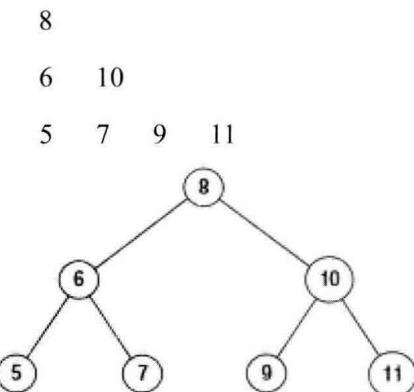


举一反三：

不管是广度优先遍历一幅有向图还是一棵树，都要用到队列。首先把起始节点（对树而言是根节点）放入队列。接下来每次从队列的头部取出一个节点，遍历这个节点之后把它能到达的节点（对树而言是子节点）都依次放入队列。重复这个遍历过程，直到队列中的节点全部被遍历为止。

题目二：分行从上到下打印二叉树。

从上到下按层打印二叉树，同一层的节点按从左到右的顺序打印，每一层打印到一行。例如，打印图4.7中二叉树的结果为：



4.7 一棵有3层的二叉树

这道面试题和前面的面试题类似，也可以用一个队列来保存将要打印的节点。为了把二叉树的每一行单独打印到一行里，我们需要两个变量：一个变量表示在当前层中还没有打印的节点数；另一个变量表示下一层节点的数目。参考代码如下：

```

void Print(BinaryTreeNode* pRoot)
{
    if(pRoot == nullptr)
        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->left != nullptr)
            nodes.push(pNode->left);
        if(pNode->right != nullptr)
            nodes.push(pNode->right);
        toBePrinted--;
        if(toBePrinted == 0)
        {
            printf("\n");
            nextLevel++;
            toBePrinted = nodes.size();
        }
    }
}
  
```

```

if(pNode->m_pLeft != nullptr)
{
    nodes.push(pNode->m_pLeft);
    ++nextLevel;
}
if(pNode->m_pRight != nullptr)
{
    nodes.push(pNode->m_pRight);
    ++nextLevel;
}
nodes.pop();
--toBePrinted;
if(toBePrinted == 0)
{
    printf("\n");
    toBePrinted = nextLevel;
    nextLevel = 0;
}
}
}

```

在上述代码中，变量 `toBePrinted` 表示在当前层中还没有打印的节点数，而变量 `nextLevel` 表示下一层的节点数。如果一个节点有子节点，则每把一个子节点加入队列，同时把变量 `nextLevel` 加 1。每打印一个节点，`toBePrinted` 减 1。当 `toBePrinted` 变成 0 时，表示当前层的所有节点已经打印完毕，可以继续打印下一层。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/32_02_PrintTreesInLines



测试用例：

- 功能测试（完全二叉树；所有节点只有左子树的二叉树；所有节点只有右子树的二叉树）。
- 特殊输入测试（二叉树根节点为 `nullptr` 指针；只有一个节点的二叉树）。



本题考点：

- 考查应聘者的思维能力。按层从上到下遍历二叉树，这对很多应聘者来说是一个新概念，要在短时间内想明白遍历的过程不是一件容易的事情。应聘者通过具体的例子找出其中的规律并想到基于队列的算法，是解决这个问题的关键。
- 考查应聘者对二叉树及队列的理解。

题目三：之字形打印二叉树。

请实现一个函数按照之字形顺序打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右到左的顺序打印，第三行再按照从左到右的顺序打印，其他行以此类推。例如，按之字形顺序打印图4.8中二叉树的结果为：

```

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

```

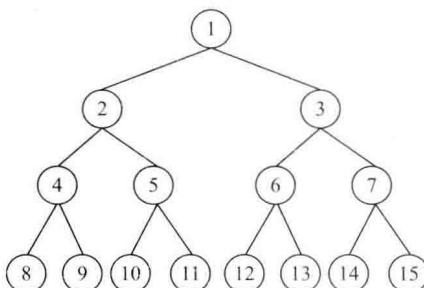


图4.8 一棵有4层的二叉树

按之字形顺序打印二叉树的过程比较复杂。如果应聘者在短时间内找不到解决办法，则一个建议是可以试着用具体的例子一步步分析。下面我们将以图4.8中的二叉树为例。

当二叉树的根节点（节点1）打印之后，它的左子节点（节点2）和右子节点（节点3）先后保存到一个数据容器里。值得注意的是，在打印第二

层的节点时，先打印节点 3，再打印节点 2。看起来节点在这个数据容器里是后进先出的，因此这个数据容器可以用栈来实现。

接着打印第二层的两个节点。根据之字形顺序，先打印节点 3，再打印节点 2，并把它们的子节点保存到一个数据容器里。我们注意到在打印第三层的时候，先打印节点 2 的两个子节点（节点 4 和节点 5），再打印节点 3 的两个子节点（节点 6 和节点 7）。这意味着我们仍然可以用一个栈来保存节点 2 和节点 3 的子节点。

我们还注意到第三层的节点是从左向右打印的，这意味着节点 4 在节点节点 5 之前打印，节点 6 在节点 7 之前打印。按照栈的后进先出特点，应该先把节点 7 保存到栈里，接着保存节点 6，再接下来是节点 5 和节点 4。也就是说，在打印第二层节点的时候，我们先把右子节点保存到栈里，再把左子节点保存到栈里。保存子节点的顺序和打印第一层时不一样。

接下来打印第三层的节点。和先前一样，在打印第三层节点的同时，我们要把第四层的节点保存到一个栈里。由于第四层的打印顺序是从右到左，因此保存的顺序是先保存左子节点，再保存右子节点。这和保存第一层根节点的两个子节点的顺序相同。

分析到这里，我们可以总结出规律：按之字形顺序打印二叉树需要两个栈。我们在打印某一层的节点时，把下一层的子节点保存到相应的栈里。如果当前打印的是奇数层（第一层、第三层等），则先保存左子节点再保存右子节点到第一个栈里；如果当前打印的是偶数层（第二层、第四层等），则先保存右子节点再保存左子节点到第二个栈里。

你可能会疑惑为什么需要两个栈。我们看看如果只有一个栈会有什么问题。在打印根节点的时候，先后把节点 2 和节点 3 保存到栈里。接下来打印节点 3。在打印节点 3 时，把它的两个子节点 6 和 7 保存到栈里。此时由于节点 7 位于栈顶，接下来将打印节点 7，而不是节点 2。为了避免这个问题，节点 6 和节点 7 要保存到另一个栈里。

表 4.5 概括了按之字形顺序打印图 4.8 中二叉树的最初 7 步，以及两个栈里保存的节点。接下来逐个把位于栈顶的节点弹出栈并打印。

表 4.5 按之字形顺序打印图 4.8 中二叉树的最初 7 步。Stack1 和 Stack2 中最右边的节点位于栈顶

步 骤	操 作	Stack1 中的节点	Stack2 中的节点
1	打印节点 1	2, 3	
2	打印节点 3	2	7, 6
3	打印节点 2		7, 6, 5, 4
4	打印节点 4	8, 9	7, 6, 5
5	打印节点 5	8, 9, 10, 11	7, 6
6	打印节点 6	8, 9, 10, 11, 12, 13	7
7	打印节点 7	8, 9, 10, 11, 12, 13, 14, 15	

分析清楚打印的流程之后，就可以动手写代码了。下面是参考代码：

```
void Print(BinaryTreeNode* pRoot)
{
    if(pRoot == nullptr)
        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 != nullptr)
                levels[next].push(pNode->m_pLeft);
            if(pNode->m_pRight != nullptr)
                levels[next].push(pNode->m_pRight);
        }
        else
        {
            if(pNode->m_pRight != nullptr)
                levels[next].push(pNode->m_pRight);
            if(pNode->m_pLeft != nullptr)
                levels[next].push(pNode->m_pLeft);
        }

        if(levels[current].empty())
        {
    }
```

```
    printf("\n");
    current = 1 - current;
    next = 1 - next;
}
}
```

上述代码定义了两个栈 `levels[0]` 和 `levels[1]`。在打印一个栈里的节点时，它的子节点保存到另一个栈里。当一层所有节点都打印完毕时，交换这两个栈并继续打印下一层。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/32_03_PrintTreesInZigzag



测试用例·

- 功能测试（完全二叉树；所有节点只有左子树的二叉树；所有节点只有右子树的二叉树）。
 - 特殊输入测试（二叉树根节点为 `nullptr` 指针；只有一个节点的二叉树）。



本题考点：

- 考查应聘者的思维能力。按之字形遍历二叉树，这对很多应聘者来说是一个新概念，要在短时间内想明白遍历的过程不是一件容易的事情。应聘者通过具体的例子找出其中的规律并想到基于两个栈的算法，是解决这个问题的关键。
 - 考查应聘者对二叉树及栈的理解。

面试题 33：二叉搜索树的后序遍历序列

题目：输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。如果是则返回 true，否则返回 false。假设输入的数组的任意两个

数字都互不相同。例如，输入数组{5, 7, 6, 9, 11, 10, 8}，则返回 true，因为这个整数序列是图 4.9 二叉搜索树的后序遍历结果。如果输入的数组是{7, 4, 6, 5}，则由于没有哪棵二叉搜索树的后序遍历结果是这个序列，因此返回 false。

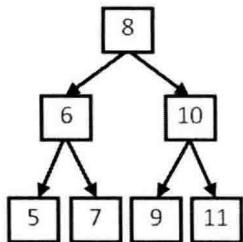


图 4.9 后序遍历序列{5,7,6,9,11,10,8}对应的二叉搜索树

在后序遍历得到的序列中，最后一个数字是树的根节点的值。数组中前面的数字可以分为两部分：第一部分是左子树节点的值，它们都比根节点的值小；第二部分是右子树节点的值，它们都比根节点的值大。

以数组{5, 7, 6, 9, 11, 10, 8}为例，后序遍历结果的最后一个数字 8 就是根节点的值。在这个数组中，前 3 个数字 5、7 和 6 都比 8 小，是值为 8 的节点的左子树节点；后 3 个数字 9、11 和 10 都比 8 大，是值为 8 的节点的右子树节点。

我们接下来用同样的方法确定与数组每一部分对应的子树的结构。这其实就是一个递归的过程。对于序列{5,7,6}，最后一个数字 6 是左子树的根节点的值。数字 5 比 6 小，是值为 6 的节点的左子节点，而 7 则是它的右子节点。同样，在序列{9,11,10}中，最后一个数字 10 是右子树的根节点，数字 9 比 10 小，是值为 10 的节点的左子节点，而 11 则是它的右子节点。

我们再来分析另一个整数数组{7, 4, 6, 5}。后序遍历的最后一个数字是根节点，因此根节点的值是 5。由于第一个数字 7 大于 5，因此在对应的二叉搜索树中，根节点上是没有左子树的，数字 7、4 和 6 都是右子树节点的值。但我们发现在右子树中有一个节点的值是 4，比根节点的值 5 小，这违背了二叉搜索树的定义。因此，不存在一棵二叉搜索树，它的后序遍历结果是{7,4,6,5}。

找到了规律之后再写代码，就不是一件很困难的事情了。下面是参考代码：

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

    int root = sequence[length - 1];

    // 在二叉搜索树中左子树节点的值小于根节点的值
    int i = 0;
    for(; i < length - 1; ++ i)
    {
        if(sequence[i] > root)
            break;
    }

    // 在二叉搜索树中右子树节点的值大于根节点的值
    int j = i;
    for(; j < length - 1; ++ j)
    {
        if(sequence[j] < root)
            return false;
    }

    // 判断左子树是不是二叉搜索树
    bool left = true;
    if(i > 0)
        left = VerifySequenceOfBST(sequence, i);

    // 判断右子树是不是二叉搜索树
    bool right = true;
    if(i < length - 1)
        right = VerifySequenceOfBST(sequence + i, length - i - 1);

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



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/33_SquenceOfBST



测试用例：

- 功能测试(输入的后序遍历序列对应一棵二叉树，包括完全二叉树、

所有节点都没有左/右子树的二叉树、只有一个节点的二叉树；输入的后序遍历序列没有对应一棵二叉树)。

- 特殊输入测试（指向后序遍历序列的指针为 nullptr 指针）。



本题考点：

- 考查应聘者分析复杂问题的思维能力。能否解决这道题的关键在于应聘者是否能找出后序遍历的规律。一旦找到了规律，用递归的代码编码相对而言就简单了。在面试的时候，应聘者可以从一两个例子入手，通过分析具体的例子寻找规律。
- 考查应聘者对二叉树后序遍历的理解。



相关题目：

输入一个整数数组，判断该数组是不是某二叉搜索树的前序遍历结果。这和前面问题的后序遍历很类似，只是在前序遍历得到的序列中，第一个数字是根节点的值。



举一反三：

如果面试题要求处理一棵二叉树的遍历序列，则可以先找到二叉树的根节点，再基于根节点把整棵树的遍历序列拆分成左子树对应的子序列和右子树对应的子序列，接下来再递归地处理这两个子序列。本面试题应用的是这种思路，面试题7“重建二叉树”应用的也是这种思路。

面试题 34：二叉树中和为某一值的路径

题目：输入一棵二叉树和一个整数，打印出二叉树中节点值的和为输入整数的所有路径。从树的根节点开始往下一直到叶节点所经过的节点形成一条路径。二叉树节点的定义如下：

```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

例如，输入图 4.10 中的二叉树和整数 22，则打印出两条路径，第一条路径包含节点 10、12，第二条路径包含节点 10、5 和 7。

一般的数据结构和算法的教材都没有介绍树的路径，因此对大多数应聘者而言，这是一个新概念，也就很难一下子想出完整的解题思路。这时候我们可以试着从一两个具体的例子入手，找到规律。

以图 4.10 中的二叉树作为例子来分析。由于路径是从根节点出发到叶节点，也就是说路径总是以根节点为起始点，因此我们首先需要遍历根节点。在树的前序、中序、后序 3 种遍历方式中，只有前序遍历是首先访问根节点的。

按照前序遍历的顺序遍历图 4.10 中的二叉树，在访问节点 10 之后，就会访问节点 5。从二叉树节点的定义可以看出，在本题的二叉树节点中没有指向父节点的指针，当访问节点 5 的时候，我们是不知道前面经过了哪些节点的，除非我们把经过的路径上的节点保存下来。每访问一个节点，我们都把当前节点添加到路径中去。当到达节点 5 时，路径中包含两个节点，它们的值分别是 10 和 5。接下来遍历到节点 4，我们把这个节点也添加到路径中。这时候已经到达叶节点，但路径上 3 个节点的值之和是 19。这个和不等于输入的值 22，因此不是符合要求的路径。

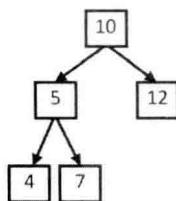


图 4.10 二叉树中有两条和为 22 的路径：一条路径经过节点 10、5、7；另一条路径经过节点 10、12

我们接着要遍历其他的节点。在遍历下一个节点之前，先要从节点 4 回到节点 5，再去遍历节点 5 的右子节点 7。值得注意的是，当回到节点 5 的时候，由于节点 4 已经不在前往节点 7 的路径上了，所以我们需要把节点 4 从路径中删除。接下来访问节点 7 的时候，再把该节点添加到路径中。此时路径中的 3 个节点 10、5、7 之和刚好是 22，是一条符合要求的路径。

我们最后要遍历的是节点 12。在遍历这个节点之前，需要先经过节点 5 回到节点 10。同样，每次当从子节点回到父节点的时候，我们都需要在

路径上删除子节点。最后在从节点 10 到达节点 12 的时候，路径上的两个节点的值之和也是 22，因此，这也是一条符合要求的路径。

我们可以用表 4.6 总结上述分析过程。

表 4.6 遍历图 4.10 中的二叉树的过程

步 骤	操 作	是否叶节点	路 径	路径节点值的和
1	访问节点 10	否	节点 10	10
2	访问节点 5	否	节点 10、节点 5	15
3	访问节点 4	是	节点 10、节点 5、节点 4	19
4	回到节点 5		节点 10、节点 5	15
5	访问节点 7	是	节点 10、节点 5、节点 7	22
6	回到节点 5		节点 10、节点 5	15
7	回到节点 10		节点 10	10
8	访问节点 12	是	节点 10、节点 12	22

分析完前面具体的例子之后，我们就找到了一些规律。当用前序遍历的方式访问到某一节点时，我们把该节点添加到路径上，并累加该节点的值。如果该节点为叶节点，并且路径中节点值的和刚好等于输入的整数，则当前路径符合要求，我们把它打印出来。如果当前节点不是叶节点，则继续访问它的子节点。当前节点访问结束后，递归函数将自动回到它的父节点。因此，我们在函数退出之前要在路径上删除当前节点并减去当前节点的值，以确保返回父节点时路径刚好是从根节点到父节点。我们不难看出保存路径的数据结构实际上是一个栈，因为路径要与递归调用状态一致，而递归调用的本质就是一个压栈和出栈的过程。

形成了清晰的思路之后，就可以动手写代码了。下面是一段参考代码：

```
void FindPath(TreeNode* pRoot, int expectedSum)
{
    if(pRoot == nullptr)
        return;

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

void FindPath
(
    TreeNode*     pRoot,
```

```

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

    // 如果是叶节点，并且路径上节点值的和等于输入的值，
    // 则打印出这条路径
    bool isLeaf = pRoot->m_pLeft == nullptr && pRoot->m_pRight == nullptr;
    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");
    }

    // 如果不是叶节点，则遍历它的子节点
    if(pRoot->m_pLeft != nullptr)
        FindPath(pRoot->m_pLeft, expectedSum, path, currentSum);
    if(pRoot->m_pRight != nullptr)
        FindPath(pRoot->m_pRight, expectedSum, path, currentSum);

    // 在返回父节点之前，在路径上删除当前节点
    path.pop_back();
}

```

在前面的代码中，我们用标准模板库中的 `vector` 实现了一个栈来保存路径，每次都用 `push_back` 在路径的末尾添加节点，用 `pop_back` 在路径的末尾删除节点，这样就保证了栈的先入后出特性。这里没有直接用 STL 中的 `stack` 的原因是，在 `stack` 中只能得到栈顶元素，而我们打印路径的时候需要得到路径上的所有节点，因此在代码实现的时候 `std::stack` 不是最好的选择。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/34_PathInTree



测试用例：

- 功能测试（二叉树中有一条、多条符合要求的路径；二叉树中没有符合要求的路径）。
- 特殊输入测试（指向二叉树根节点的指针为 `nullptr` 指针）。



本题考点：

- 考查应聘者分析复杂问题的思维能力。应聘者遇到这个问题的时候，如果一下子没有思路，则不妨从一个具体的例子开始，一步步分析路径上包含哪些节点，这样就能找出其中的规律，从而想到解决方案。
- 考查应聘者对二叉树的前序遍历的理解。

4.4

分解让复杂问题简单化

很多读者可能都知道“各个击破”的军事思想，这种思想的精髓是当敌我实力悬殊时，我们可以把强大的敌人分割开来，然后集中优势兵力打败被分割开来的小部分敌人。要一下子战胜总体很强大的敌人很困难，但战胜小股敌人就容易多了。同样，在面试中，当我们遇到复杂的大问题的时候，如果能够先把大问题分解成若干个简单的小问题，然后再逐个解决这些小问题，则可能也会容易很多。

我们可以按照解决问题的步骤来分解复杂问题，每一步解决一个小问题。比如在面试题35“复杂链表的复制”中，我们将复杂链表复制的过程分解成3个步骤。在写代码的时候我们为每一步定义一个函数，这样每个函数完成一个功能，整个过程的逻辑也就非常清晰明了了。

在计算机领域有一类算法叫分治法，即“分而治之”，采用的就是各个击破的思想。我们把分解之后的小问题各个解决，然后把小问题的解决方案结合起来解决大问题。比如在面试题36“二叉搜索树与双向链表”中，转换整棵二叉树是一个大问题，我们把这个大问题分解成转换左子树和右子树两个小问题，然后再把转换左、右子树得到的链表和根节点链接起

来，就解决了整棵大问题。通常分治法都可以用递归的代码实现。

在面试题 38 “字符串的排列”中，我们把整个字符串分为两部分：第一个字符及它后面的所有字符。我们先拿第一个字符和后面的每个字符交换，交换之后再求后面所有字符的排列。整个字符串的排列是一个大问题，而第一个字符之后的字符串的排列是一个小问题。因此，这实际上也是分治法的应用，可以用递归实现。

面试题 35：复杂链表的复制

题目：请实现函数 `ComplexListNode* Clone(ComplexListNode* pHead)`，复制一个复杂链表。在复杂链表中，每个节点除了有一个 `m_pNext` 指针指向下一个节点，还有一个 `m_pSibling` 指针指向链表中的任意节点或者 `nullptr`。节点的 C++ 定义如下：

```
struct ComplexListNode
{
    int m_nValue;
    ComplexListNode* m_pNext;
    ComplexListNode* m_pSibling;
};
```

图 4.11 是一个含有 5 个节点的复杂链表。图中实线箭头表示 `m_pNext` 指针，虚线箭头表示 `m_pSibling` 指针。为简单起见，指向 `nullptr` 的指针没有画出。

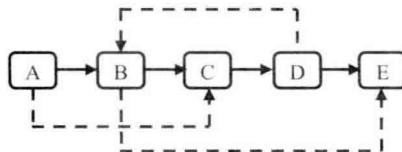


图 4.11 一个含有 5 个节点的复杂链表

注：在复杂链表的节点中，除了有指向下一个节点的指针（实线箭头），还有指向任意节点的指针（虚线箭头）。

听到这个问题之后，很多应聘者的第一反应是把复制过程分成两步：第一步是复制原始链表上的每个节点，并用 `m_pNext` 链接起来；第二步是设置每个节点的 `m_pSibling` 指针。假设原始链表中的某个节点 N 的 `m_pSibling` 指向节点 S，由于 S 在链表中可能在 N 的前面也可能在 N 的后面，所以要定位 S 的位置需要从原始链表的头节点开始找。如果从原始链

表的头节点开始沿着 `m_pNext` 经过 s 步找到节点 S ，那么在复制链表上节点 N' 的 `m_pSibling`（记为 S' ）离复制链表的头节点的距离也是沿着 `m_pNext` 指针 s 步。用这种办法就可以为复制链表上的每个节点设置 `m_pSibling` 指针。

对于一个含有 n 个节点的链表，由于定位每个节点的 `m_pSibling` 都需要从链表头节点开始经过 $O(n)$ 步才能找到，因此这种方法总的时间复杂度是 $O(n^2)$ 。

由于上述方法的时间主要花费在定位节点的 `m_pSibling` 上面，我们试着在这方面去进行优化。我们还是分为两步：第一步仍然是复制原始链表上的每个节点 N 创建 N' ，然后把这些创建出来的节点用 `m_pNext` 链接起来。同时我们把 $\langle N, N' \rangle$ 的配对信息放到一个哈希表中；第二步还是设置复制链表上每个节点的 `m_pSibling`。如果在原始链表中节点 N 的 `m_pSibling` 指向节点 S ，那么在复制链表中，对应的 N' 应该指向 S' 。由于有了哈希表，我们可以用 $O(1)$ 的时间根据 S 找到 S' 。

第二种方法相当于用空间换时间。对于有 n 个节点的链表，我们需要一个大小为 $O(n)$ 的哈希表，也就是说我们以 $O(n)$ 的空间消耗把时间复杂度由 $O(n^2)$ 降低到 $O(n)$ 。

接下来我们再换一种思路，在不用辅助空间的情况下实现 $O(n)$ 的时间效率。第三种方法的第一步仍然是根据原始链表的每个节点 N 创建对应的 N' 。这一次，我们把 N' 链接在 N 的后面。图 4.11 中的链表经过这一步之后的结构如图 4.12 所示。

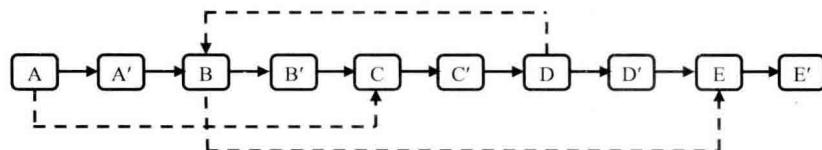


图 4.12 复制复杂链表的第一步

注：复制原始链表的任意节点 N 并创建新节点 N' ，再把 N' 链接到 N 的后面。

完成这一步的代码如下：

```
void CloneNodes(ComplexListNode* pHead)
{
    ComplexListNode* pNode = pHead;
    while(pNode != nullptr)
```

```

{
    ComplexListNode* pCloned = new ComplexListNode();
    pCloned->m_nValue = pNode->m_nValue;
    pCloned->m_pNext = pNode->m_pNext;
    pCloned->m_pSibling = nullptr;

    pNode->m_pNext = pCloned;

    pNode = pCloned->m_pNext;
}
}

```

第二步设置复制出来的节点的 `m_pSibling`。假设原始链表上的 `N` 的 `m_pSibling` 指向节点 `S`，那么其对应复制出来的 `N'` 是 `N` 的 `m_pNext` 指向的节点，同样 `S'` 也是 `S` 的 `m_pNext` 指向的节点。设置 `m_pSibling` 之后的链表如图 4.13 所示。

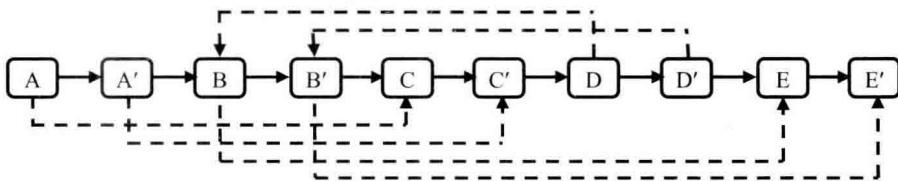


图 4.13 复制复杂链表的第二步

注：如果原始链表上的节点 `N` 的 `m_pSibling` 指向 `S`，则它对应的复制节点 `N'` 的 `m_pSibling` 指向 `S` 的复制节点 `S'`。

下面是完成第二步的参考代码：

```

void ConnectSiblingNodes(ComplexListNode* pHead)
{
    ComplexListNode* pNode = pHead;
    while(pNode != nullptr)
    {
        ComplexListNode* pCloned = pNode->m_pNext;
        if(pNode->m_pSibling != nullptr)
        {
            pCloned->m_pSibling = pNode->m_pSibling->m_pNext;
        }

        pNode = pCloned->m_pNext;
    }
}

```

第三步把这个长链表拆分成两个链表：把奇数位置的节点用 `m_pNext` 链接起来就是原始链表，把偶数位置的节点用 `m_pNext` 链接起来就是复制出来的链表。图 4.13 中的链表拆分之后的两个链表如图 4.14 所示。

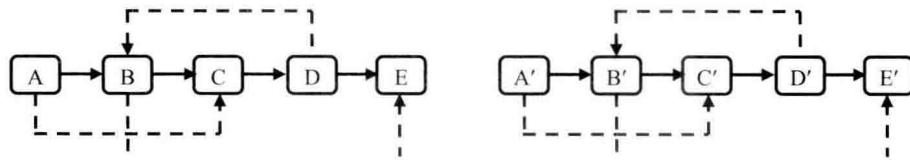


图 4.14 复制复杂链表的第三步

注：把第二步得到的链表拆分成两个链表，奇数位置上的节点组成原始链表，偶数位置上的节点组成复制出来的链表。

要实现第三步的操作也不是很难的事情。其对应的代码如下：

```
ComplexListNode* ReconnectNodes(ComplexListNode* pHead)
{
    ComplexListNode* pNode = pHead;
    ComplexListNode* pClonedHead = nullptr;
    ComplexListNode* pClonedNode = nullptr;

    if(pNode != nullptr)
    {
        pClonedHead = pClonedNode = pNode->m_pNext;
        pNode->m_pNext = pClonedNode->m_pNext;
        pNode = pNode->m_pNext;
    }

    while(pNode != nullptr)
    {
        pClonedNode->m_pNext = pNode->m_pNext;
        pClonedNode = pClonedNode->m_pNext;
        pNode->m_pNext = pClonedNode->m_pNext;
        pNode = pNode->m_pNext;
    }

    return pClonedHead;
}
```

我们把上面三步合起来，就是复制链表的完整过程。

```
ComplexListNode* Clone(ComplexListNode* pHead)
{
    CloneNodes(pHead);
    ConnectSiblingNodes(pHead);
    return ReconnectNodes(pHead);
}
```



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/35_CopyComplexList



测试用例：

- 功能测试(节点中的 m_pSibling 指向节点自身; 两个节点的 m_pSibling 形成环状结构; 链表中只有一个节点)。
- 特殊输入测试 (指向链表头节点的指针为 nullptr 指针)。



本题考点：

- 考查应聘者对复杂问题的思维能力。本题中的复杂链表是一种不太常见的数据结构，而且复制这种链表的过程也较为复杂。我们把复杂链表的复制过程分解成 3 个步骤，同时把每个步骤都用图形化的方式表示出来，这样能帮助我们厘清思路。在写代码的时候，我们为每个步骤定义一个子函数，最后在复制函数中先后调用这 3 个函数。有了这些清晰的思路之后再写代码，就容易多了。
- 考查应聘者分析时间效率和空间效率的能力。当应聘者提出第一种和第二种思路的时候，面试官会提示此时在效率上还不是最优解。这时候应聘者要能自己分析出这两种算法的时间复杂度和空间复杂度各是多少。

面试题 36：二叉搜索树与双向链表

题目：输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的节点，只能调整树中节点指针的指向。比如，输入图 4.15 中左边的二叉搜索树，则输出转换之后的排序双向链表。二叉树节点的定义如下：

```
struct BinaryTreeNode
{
    int m_nValue;
    BinaryTreeNode* m_pLeft;
    BinaryTreeNode* m_pRight;
};
```

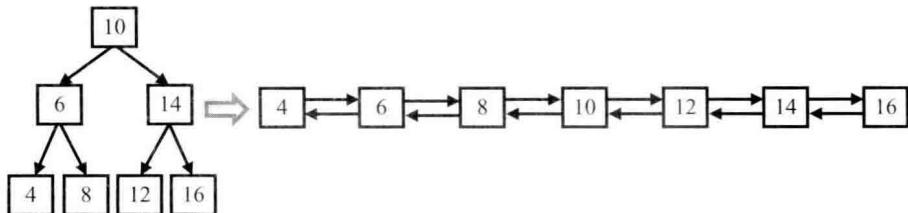


图 4.15 一棵二叉搜索树及转换之后的排序双向链表

在二叉树中，每个节点都有两个指向子节点的指针。在双向链表中，每个节点也有两个指针，分别指向前一个节点和后一个节点。由于这两种节点的结构相似，同时二叉搜索树也是一种排序的数据结构，因此，在理论上有可能实现二叉搜索树和排序双向链表的转换。在搜索二叉树中，左子节点的值总是小于父节点的值，右子节点的值总是大于父节点的值。因此，我们在将二叉搜索树转换成排序双向链表时，原先指向左子节点的指针调整为链表中指向前一个节点的指针，原先指向右子节点的指针调整为链表中指向后一个节点的指针。接下来我们考虑该如何转换。

由于要求转换之后的链表是排好序的，我们可以中序遍历树中的每个节点，这是因为中序遍历算法的特点是按照从小到大的顺序遍历二叉树的每个节点。当遍历到根节点的时候，我们把树看成3部分：值为10的节点；根节点值为6的左子树；根节点值为14的右子树。根据排序链表的定义，值为10的节点将和它的左子树的最大一个节点（值为8的节点）链接起来，同时它还将和右子树最小的节点（值为12的节点）链接起来，如图4.16所示。

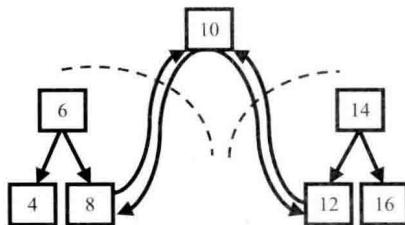


图 4.16 把二叉搜索树看成3部分

注：根节点、左子树和右子树。在把左、右子树都转换成排序双向链表之后再和根节点链接起来，整棵二叉搜索树也就转换成了排序双向链表。

按照中序遍历的顺序，当我们遍历转换到根节点（值为 10 的节点）时，它的左子树已经转换成一个排序的链表了，并且处在链表中的最后一个节点是当前值最大的节点。我们把值为 8 的节点和根节点链接起来，此时链表中的最后一个节点就是 10 了。接着我们去遍历转换右子树，并把根节点和右子树中最小的节点链接起来。至于怎么去转换它的左子树和右子树，由于遍历和转换过程是一样的，我们很自然地想到可以用递归。

基于上述分析过程，我们可以写出如下代码：

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

    // pLastNodeInList 指向双向链表的尾节点,
    // 我们需要返回头节点
    BinaryTreeNode *pHeadOfList = pLastNodeInList;
    while(pHeadOfList != nullptr && pHeadOfList->m_pLeft != nullptr)
        pHeadOfList = pHeadOfList->m_pLeft;

    return pHeadOfList;
}

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

    BinaryTreeNode *pCurrent = pNode;

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

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

    *pLastNodeInList = pCurrent;

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

在上面的代码中，我们用 pLastNodeInList 指向已经转换好的链表的最后一个节点（值最大的节点）。当我们遍历到值为 10 的节点的时候，它的左子树都已经转换好了，因此 pLastNodeInList 指向值为 8 的节点。接着把根节点链接到链表中之后，值为 10 的节点成了链表中的最后一个节点（新

的值最大的节点），于是 pLastNodeInList 指向了这个值为 10 的节点。接下来把 pLastNodeInList 作为参数传入函数递归遍历右子树。我们找到右子树中最左边的子节点（值为 12 的节点，在右子树中值最小），并把该节点和值为 10 的节点链接起来。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/36_ConvertBinarySearchTree



测试用例：

- 功能测试（输入的二叉树是完全二叉树；所有节点都没有左/右子树的二叉树；只有一个节点的二叉树）。
- 特殊输入测试（指向二叉树根节点的指针为 nullptr 指针）。



本题考点：

- 考查应聘者分析复杂问题的能力。无论是二叉树还是双向链表，都有很多指针。要实现这两种不同数据结构的转换，需要调整大量的指针，因此这个过程会很复杂。为了把这个复杂的问题分析清楚，我们可以把树分为 3 部分：根节点、左子树和右子树，然后把左子树中最大的节点、根节点、右子树中最小的节点链接起来。至于如何把左子树和右子树内部的节点链接成链表，那和原来的问题的实质是一样的，因此可以递归解决。解决这个问题的关键在于把一个大的问题分解成几个小问题，并递归地解决小问题。
- 考查应聘者对二叉树和双向链表的理解及编程能力。

面试题 37：序列化二叉树

题目：请实现两个函数，分别用来序列化和反序列化二叉树。

通过分析解决面试题 7 “重建二叉树”，我们知道可以从前序遍历序列

和中序遍历序列中构造出一棵二叉树。受此启发，我们可以先把一棵二叉树序列化成一个前序遍历序列和一个中序遍历序列，然后在反序列化时通过这两个序列重构出原二叉树。

这种思路有两个缺点：一是该方法要求二叉树中不能有数值重复的节点；二是只有当两个序列中所有数据都读出后才能开始反序列化。如果两个遍历序列的数据是从一个流里读出来的，那么可能需要等待较长的时间。

实际上，如果二叉树的序列化是从根节点开始的，那么相应的反序列化在根节点的数值读出来的时候就可以开始了。因此，我们可以根据前序遍历的顺序来序列化二叉树，因为前序遍历是从根节点开始的。在遍历二叉树碰到 `nullptr` 指针时，这些 `nullptr` 指针序列化为一个特殊的字符（如'\$'）。另外，节点的数值之间要用一个特殊字符（如','）隔开。

根据这样的序列化规则，图 4.17 中的二叉树被序列化成字符串 "1,2,4,\$,\$,\$,3,5,\$,\$,6,\$,\$"。

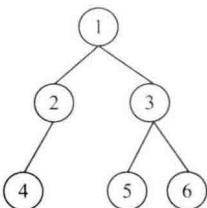


图 4.17 一棵被序列化成字符串"1,2,4,\$,\$,\$,3,5,\$,\$,6,\$,\$"的二叉树

上述序列化过程可以用如下 C++ 代码实现：

```

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

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

我们接着以字符串"1,2,4,\$,\$,\$,3,5,\$,\$,6,\$,\$"为例分析如何反序列化二叉树。第一个读出的数字是 1。由于前序遍历是从根节点开始的，这是根节点的值。接下来读出的数字是 2，根据前序遍历的规则，这是根节点的左子节点的值。同样，接下来的数字 4 是值为 2 的节点的左子节点。接着从序列

化字符串里读出两个字符'\$'，这表明值为4的节点的左、右子节点均为`nullptr`指针，因此它是一个叶节点。接下来回到值为2的节点，重建它的右子节点。由于下一个字符是'\$'，这表明值为2的节点的右子节点为`nullptr`指针。这个节点的左、右子树都已经构建完毕，接下来回到根节点，反序列化根节点的右子树。

下一个序列化字符串中的数字是3，因此右子树的根节点的值为3。它的左子节点是一个值为5的叶节点，因为接下来的三个字符是"5,\$,\$"。同样，它的右子节点是值为6的叶节点，因为最后3个字符是"6,\$,\$"。

上述反序列化过程可以用如下C++代码实现：

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

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

函数`ReadStream`每次从流中读出一个数字或者一个字符'\$'。当从流中读出的是一个数字时，函数返回`true`；否则返回`false`。

如果总结前面序列化和反序列化的过程，就会发现我们都是把二叉树分解成3部分：根节点、左子树和右子树。我们在处理（序列化或反序列化）它的根节点之后再分别处理它的左、右子树。这是典型的把问题递归分解然后逐个解决的过程。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/37_SerializeBinaryTrees



测试用例：

- 功能测试（输入的二叉树是完全二叉树；所有节点都没有左/右子树的二叉树；只有一个节点的二叉树；所有节点的值都相同的二叉树）。
- 特殊输入测试（指向二叉树根节点的指针为 nullptr 指针）。



本题考点：

- 考查应聘者分析复杂问题的能力。为了把这个问题分析清楚，我们可以把树分为 3 部分：根节点、左子树和右子树，在序列化/反序列化根节点之后再分别序列化/反序列化左、右子树，因此可以递归解决。解决这个问题的关键在于把一个大的问题分解成几个小问题，并递归地解决小问题。
- 考查应聘者对二叉树遍历的理解及编程能力。

面试题 38：字符串的排列

题目：输入一个字符串，打印出该字符串中字符的所有排列。例如，输入字符串 abc，则打印出由字符 a、b、c 所能排列出来的所有字符串 abc、acb、bac、bca、cab 和 cba。

如何求出几个字符的所有排列，很多人都不能一下子想出解决方案。那我们是不是可以考虑把这个复杂的问题分解成小的问题呢？比如，我们把一个字符串看成由两部分组成：第一部分是它的第一个字符；第二部分是后面的所有字符。在图 4.18 中，我们用两种不同的背景颜色区分字符串的两部分。

我们求整个字符串的排列，可以看成两步。第一步求所有可能出现在第一个位置的字符，即把第一个字符和后面所有的字符交换。图 4.18 就是分别把第一个字符 a 和后面的 b、c 等字符交换的情形。第二步固定第一个字符，如图 4.18 (a) 所示，求后面所有字符的排列。这时候我们仍把后面的所有字符分成两部分：后面字符的第一个字符，以及这个字符之后的所有字符。然后把第一个字符逐一和它后面的字符交换，如图 4.18 (b) 所示。

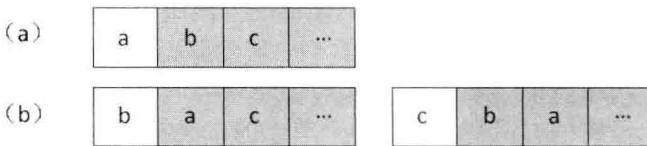


图 4.18 求字符串的排列的过程

注：(a) 把字符串分为两部分：一部分是字符串的第一个字符；另一部分是第一个字符以后的所有字符（有阴影背景的区域）。接下来求阴影部分的字符串的排列。(b) 拿第一个字符和它后面的字符逐个交换。

分析到这里我们就可以看出，这其实是典型的递归思路，于是不难写出如下代码：

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

    Permutation(pStr, pStr);
}

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

            Permutation(pStr, pBegin + 1);

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

在函数 Permutation(char* pStr, char* pBegin) 中，指针 pStr 指向整个字符串的第一个字符，pBegin 指向当前我们执行排列操作的字符串的第一个字符。在每一次递归的时候，我们从 pBegin 向后扫描每一个字符（指针 pCh

指向的字符)。在交换 pBegin 和 pCh 指向的字符之后，我们再对 pBegin 后面的字符串递归地进行排列操作，直至 pBegin 指向字符串的末尾。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/38_StringPermutation



测试用例：

- 功能测试（输入的字符串中有一个或者多个字符）。
- 特殊输入测试（输入的字符串的内容为空或者 `nullptr` 指针）。



本题考点：

- 考查应聘者的思维能力。当整个问题看起来不能直接解决的时候，应聘者能否想到把字符串分成两部分，从而把大问题分解成小问题来解决，是能否顺利解决这个问题的关键。
- 考查应聘者对递归的理解和编程能力。



本题扩展：

如果不是求字符的所有排列，而是求字符的所有组合，应该怎么办呢？还是输入三个字符 a、b、c，则它们的组合有 a、b、c、ab、ac、bc、abc。当交换字符串中的两个字符时，虽然能得到两个不同的排列，但却是同一个组合。比如 ab 和 ba 是不同的排列，但只算一个组合。

如果输入 n 个字符，则这 n 个字符能构成长度为 $1, 2, \dots, n$ 的组合。在求 n 个字符的长度为 m ($1 \leq m \leq n$) 的组合的时候，我们把这 n 个字符分成两部分：第一个字符和其余的所有字符。如果组合里包含第一个字符，则下一步在剩余的字符串里选取 $m-1$ 个字符；如果组合里不包含第一个字符，则下一步在剩余的 $n-1$ 个字符串里选取 m 个字符。也就是说，我们可以把求 n 个字符组成长度为 m 的组合的问题分解成两个子问题，分别求 $n-1$ 个字符

中长度为 $m-1$ 的组合，以及求 $n-1$ 个字符串中长度为 m 的组合。这两个子问题都可以用递归的方式解决。



相关题目：

- 输入一个含有 8 个数字的数组，判断有没有可能把这 8 个数字分别放到正方体的 8 个顶点上（如图 4.19 所示），使得正方体上三组相对的面上的 4 个顶点的和都相等。

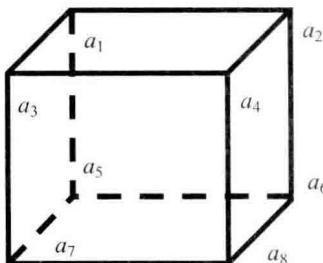


图 4.19 把 8 个数字放到正方体的 8 个顶点上

这相当于先得到 a_1 、 a_2 、 a_3 、 a_4 、 a_5 、 a_6 、 a_7 和 a_8 这 8 个数字的所有排列，然后判断有没有某一个排列符合题目给定的条件，即 $a_1+a_2+a_3+a_4=a_5+a_6+a_7+a_8$ ， $a_1+a_3+a_5+a_7=a_2+a_4+a_6+a_8$ ，并且 $a_1+a_2+a_5+a_6=a_3+a_4+a_7+a_8$ 。

- 在 8×8 的国际象棋上摆放 8 个皇后，使其不能相互攻击，即任意两个皇后不得处在同一行、同一列或者同一条对角线上。图 4.20 中的每个黑色格子表示一个皇后，这就是一种符合条件的摆放方法。请问总共有多少种符合条件的摆法？

由于 8 个皇后的任意两个不能处在同一行，那么肯定是每一个皇后占据一行。于是我们可以定义一个数组 `ColumnIndex[8]`，数组中第 i 个数字表示位于第 i 行的皇后的列号。先把数组 `ColumnIndex` 的 8 个数字分别用 0~7 初始化，然后对数组 `ColumnIndex` 进行全排列。因为我们用不同的数字初始化数组，所以任意两个皇后肯定不同列。只需判断每一个排列对应的 8 个皇后是不是在同一条对角线上，也就是对于数组的两个下标 i 和 j ，是否有 $i-j==ColumnIndex[i]-ColumnIndex[j]$ 或者 $j-i==ColumnIndex[i]-ColumnIndex[j]$ 。

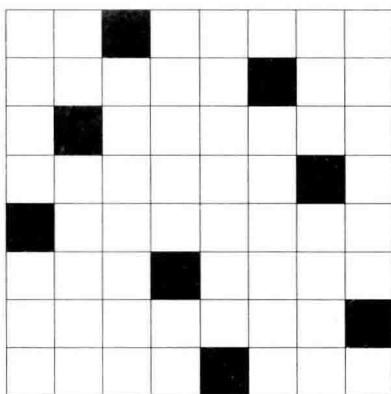


图 4.20 8×8 的国际象棋棋盘上摆着 8 个皇后（黑色小方格），任意两个皇后不在同一行、同一列或者同一条对角线上



举一反三：

如果面试题是按照一定要求摆放若干个数字，则可以先求出这些数字的所有排列，然后一一判断每个排列是不是满足题目给定的要求。

4.5

本章小结

在面试时，我们难免会遇到难题，画图、举例和分解这 3 种方法能够帮助我们解决复杂的问题，如图 4.21 所示。

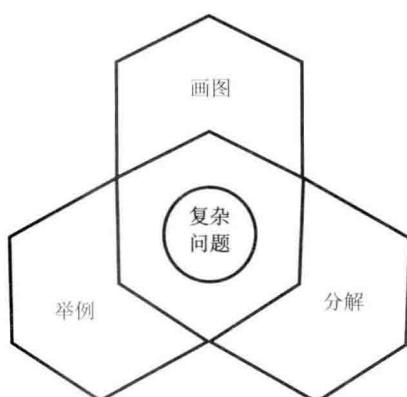


图 4.21 解决复杂问题的 3 种方法：画图、举例和分解

图形能使抽象的问题形象化。当面试题涉及链表、二叉树等数据结构时，如果在纸上画几张草图，则题目中隐藏的规律就有可能变得很直观。

一两个例子能使抽象的问题具体化。很多与算法相关的问题都很抽象，未必一眼就能看出它们的规律。这时候我们不妨举几个例子，一步一步模拟运行的过程，说不定就能发现其中的规律，从而找到解决问题的窍门。

把复杂问题分解成若干个小问题，是解决很多复杂问题的有效方法。如果我们遇到的问题很大，则可以尝试先把大问题分解成小问题，然后再递归地解决这些小问题。分治法、动态规划等方法应用的都是分解复杂问题的思路。

第 5 章

优化时间和空间效率

5.1 面试官谈效率

“通常针对有较长工作经验的应聘者，会问一些关于时间和空间效率的问题，这能够体现一个应聘者较好的编程素质和能力。”

——刘景勇（Autodesk，软件工程师）

“面试时一般会直接要求空间和时间复杂度，这两者都很重要。”

——张珺（百度，高级软件工程师）

“我们有很多考查时间和空间效率这方面的问题。通常两者都给应聘者限定，然后让他给出解决方案。”

——张晓禹（百度，技术经理）

“只要不是特别大的内存开销，时间复杂度比较重要。因为改进时间复杂度对算法的要求更高。”

——吴斌（英伟达，图形设计师）

“是空间换时间还是时间换空间，这要看具体的题目了。对于普通的应用，一般是空间换时间，因为通常用户更关心速度，而且一般有足够的存储空间允许这么做。但对于现在的一般嵌入式设备，很多时候空间换时间就不现实了，因为存储空间太少了。”

——陈黎明（微软，SDE II）

5.2 时间效率

由于每个人都希望软件的响应时间尽量短一些，所以软件公司都很重视软件的时间性能，都会在发布软件之前花不少精力进行时间效率优化。这也就不难理解为什么很多公司的面试官都把代码的时间效率当作一个考查重点。面试官除了考查应聘者的编程能力，还关注应聘者有没有不断优化效率、追求完美的态度和能力。

首先，我们的编程习惯对代码的时间效率有很大影响。比如 C/C++程序员要养成采用引用（或指针）传递复杂类型参数的习惯。如果采用值传递的方式，则从形参到实参会产生一次复制操作。这样的复制是多余的操作，我们应该尽量避免。再举个例子，如果用 C#做多次字符串的拼接操作，则不要多次用 String 的+运算符来拼接字符串，因为这样会产生很多 String 的临时实例，造成时间和空间的浪费。更好的办法是用 StringBuilder 的 Append 方法来完成字符串的拼接。如果我们平时不太注意这些影响代码效率的细节，没有养成好的编码习惯，那么我们写出的代码可能会让面试官大失所望。

其次，即使同一个算法用循环和递归两种思路实现的时间效率可能会大不一样。递归的本质是把一个大的复杂问题分解成两个或者多个小的简单问题。如果小问题中有相互重叠的部分，那么直接用递归实现虽然代码

显得很简洁，但时间效率可能会非常差（详细讨论见本书 2.4.1 节）。对于这种类型的题目，我们可以用递归的思路来分析问题，但写代码的时候可以用数组（一维或者多维数组）来保存中间结果基于循环实现。绝大部分动态规划算法的分析和代码实现都是分这两个步骤完成的。详细的讨论请参考面试题 47 “礼物的最大价值” 和面试题 48 “最长不含重复字符的子字符串”。

再次，代码的时间效率还能体现应聘者对数据结构和算法功底的掌握程度。同样是查找，如果是顺序查找则需要 $O(n)$ 的时间；如果输入的是排序的数组则只需要 $O(\log n)$ 的时间；如果事先已经构造好了哈希表，那么查找在 $O(1)$ 时间内就能完成。我们只有对常见的数据结构和算法都了然于胸，才能在需要的时候选择合适的数据结构和算法来解决问题。

最后，应聘者在面试的时候要展示敏捷的思维能力和追求完美的激情。听到题目的时候，我们一般很快就能想到最直观的算法。这个最直观的办法很有可能不是最优的，但也不妨在第一时间告诉面试官，这样面试官至少会觉得我们思维比较敏捷。我们想到几种思路之后，面试官可能仍然不满意，还在提示我们有更好的办法。这时候我们一定不能轻言放弃，而要表现出积极思考的态度，努力从不同的角度去思考问题。有些题目很难，面试官甚至不期待应聘者在短短几十分钟里想出完美的解法，但他会希望应聘者能够有激情、有耐心地去尝试新的思路，而不是碰到难题就退缩。在面试的时候，应聘者的态度和激情对最终的面试结果也有很重要的影响。

面试题 39：数组中出现次数超过一半的数字

题目：数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如，输入一个长度为 9 的数组 {1, 2, 3, 2, 2, 2, 5, 4, 2}。由于数字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2。

看到这道题，很多应聘者就会想要是这个数组是排序的数组就好了。如果是排好序的数组，那么我们就能很容易统计出每个数字出现的次数。题目给出的数组没有说是排序的，因此我们需要先给它排序。排序的时间复杂度是 $O(n \log n)$ 。最直观的算法通常不是面试官满意的算法，接下来我们试着找出更快的算法。

❖ 解法一：基于 Partition 函数的时间复杂度为 $O(n)$ 的算法

如果我们回到题目本身仔细分析，就会发现前面的思路并没有考虑到数组的特性：数组中有一个数字出现的次数超过了数组长度的一半。如果把这个数组排序，那么排序之后位于数组中间的数字一定就是那个出现次数超过数组长度一半的数字。也就是说，这个数字就是统计学上的中位数，即长度为 n 的数组中第 $n/2$ 大的数字。我们有成熟的时间复杂度为 $O(n)$ 的算法得到数组中任意第 k 大的数字。

这种算法受快速排序算法的启发。在随机快速排序算法中，我们先在数组中随机选择一个数字，然后调整数组中数字的顺序，使得比选中的数字小的数字都排在它的左边，比选中的数字大的数字都排在它的右边。如果这个选中的数字的下标刚好是 $n/2$ ，那么这个数字就是数组的中位数；如果它的下标大于 $n/2$ ，那么中位数应该位于它的左边，我们可以接着在它的左边部分的数组中查找；如果它的下标小于 $n/2$ ，那么中位数应该位于它的右边，我们可以接着在它的右边部分的数组中查找。这是一个典型的递归过程，可以用如下代码实现：

```
int MoreThanHalfNum(int* numbers, int length)
{
    if(CheckInvalidArray(numbers, length))
        return 0;

    int middle = length >> 1;
    int start = 0;
    int end = length - 1;
    int index = Partition(numbers, length, start, end);
    while(index != middle)
    {
        if(index > middle)
        {
            end = index - 1;
            index = Partition(numbers, length, start, end);
        }
        else
        {
            start = index + 1;
            index = Partition(numbers, length, start, end);
        }
    }

    int result = numbers[middle];
    if(!CheckMoreThanHalf(numbers, length, result))
        result = 0;
}
```

```

    return result;
}

```

上述代码中的函数 Partition 是完成快速排序的基础。我们在本书的 2.4.2 节详细讨论了这个函数，这里不再重复。

在面试的时候，除了要完成基本功能即找到符合要求的数字，还要考虑一些无效的输入。如果函数的输入参数是一个指针（数组在参数传递的时候退化为指针），就要考虑这个指针可能为 `nullptr`。下面的函数 `CheckInvalidArray` 用来判断输入的数组是不是无效的。题目中说数组中有一个数字出现的次数超过数组长度的一半，如果输入的数组中出现频率最高的数字都没有达到这个标准，那该怎么办？这就是我们定义了一个 `CheckMoreThanHalf` 函数的原因。面试的时候我们要全面考虑这些情况，才能让面试官完全满意。下面的代码用一个全局变量来表示输入无效的情况。更多关于出错处理的讨论，详见本书 3.3 节。

```

bool g_bInputInvalid = false;

bool CheckInvalidArray(int* numbers, int length)
{
    g_bInputInvalid = false;
    if(numbers == nullptr || length <= 0)
        g_bInputInvalid = true;

    return g_bInputInvalid;
}

bool CheckMoreThanHalf(int* numbers, int length, int number)
{
    int times = 0;
    for(int i = 0; i < length; ++i)
    {
        if(numbers[i] == number)
            times++;
    }

    bool isMoreThanHalf = true;
    if(times * 2 <= length)
    {
        g_bInputInvalid = true;
        isMoreThanHalf = false;
    }

    return isMoreThanHalf;
}

```

❖ 解法二：根据数组特点找出时间复杂度为 $O(n)$ 的算法

接下来我们从另外一个角度来解决这个问题。数组中有一个数字出现的次数超过数组长度的一半，也就是说它出现的次数比其他所有数字出现次数的和还要多。因此，我们可以考虑在遍历数组的时候保存两个值：一个是数组中的一个数字；另一个是次数。当我们遍历到下一个数字的时候，如果下一个数字和我们之前保存的数字相同，则次数加1；如果下一个数字和我们之前保存的数字不同，则次数减1。如果次数为零，那么我们需要保存下一个数字，并把次数设为1。由于我们要找的数字出现的次数比其他所有数字出现的次数之和还要多，那么要找的数字肯定是最后一次把次数设为1时对应的数字。

下面是这种思路的参考代码：

```
int MoreThanHalfNum(int* numbers, int length)
{
    if(CheckInvalidArray(numbers, length))
        return 0;

    int result = numbers[0];
    int times = 1;
    for(int i = 1; i < length; ++i)
    {
        if(times == 0)
        {
            result = numbers[i];
            times = 1;
        }
        else if(numbers[i] == result)
            times++;
        else
            times--;
    }

    if(!CheckMoreThanHalf(numbers, length, result))
        result = 0;

    return result;
}
```

和第一种思路一样，我们也要检验输入的数组是不是有效的，这里不再重复。

❖ 解法比较

上述两种算法的时间复杂度都是 $O(n)$ 。基于 Partition 函数的算法的时

间复杂度的分析不是很直观，本书限于篇幅不作详细讨论，感兴趣的读者可以参考《算法导论》等书籍的相关章节。我们注意到，在第一种解法中，需要交换数组中数字的顺序，这就会修改输入的数组。是不是可以修改输入的数组呢？在面试的时候，我们可以和面试官讨论，让他明确需求。如果面试官说不能修改输入的数组，那就只能采用第二种解法了。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/39_MoreThanHalfNumber



测试用例：

- 功能测试（输入的数组中存在一个出现次数超过数组长度一半的数字；输入的数组中不存在一个出现次数超过数组长度一半的数字）。
- 特殊输入测试（输入的数组中只有一个数字；输入 `nullptr` 指针）。



本题考点：

- 考查应聘者对时间复杂度的理解。应聘者每想出一种解法，面试官都期待他能分析出这种解法的时间复杂度是多少。
- 考查应聘者思维的全面性。面试官除了要求应聘者能对有效的输入返回正确的结果，同时也期待应聘者能对无效的输入进行相应的处理。

面试题 40：最小的 k 个数

题目：输入 n 个整数，找出其中最小的 k 个数。例如，输入 4、5、1、6、2、7、3、8 这 8 个数字，则最小的 4 个数字是 1、2、3、4。

这道题最简单的思路莫过于把输入的 n 个整数排序，排序之后位于最前面的 k 个数就是最小的 k 个数。这种思路的时间复杂度是 $O(n \log n)$ ，面试官会提示我们还有更快的算法。

❖ 解法一：时间复杂度为 $O(n)$ 的算法，只有当我们可以修改输入的数组时可用

从解决面试题 39 “数组中出现次数超过一半的数字”得到了启发，我们同样可以基于 Partition 函数来解决这个问题。如果基于数组的第 k 个数字来调整，则使得比第 k 个数字小的所有数字都位于数组的左边，比第 k 个数字大的所有数字都位于数组的右边。这样调整之后，位于数组中左边的 k 个数字就是最小的 k 个数字（这 k 个数字不一定是排序的）。下面是基于这种思路的参考代码：

```
void GetLeastNumbers(int* input, int n, int* output, int k)
{
    if(input == nullptr || output == nullptr || k > n || n <= 0 || k <= 0)
        return;

    int start = 0;
    int end = n - 1;
    int index = Partition(input, n, start, end);
    while(index != k - 1)
    {
        if(index > k - 1)
        {
            end = index - 1;
            index = Partition(input, n, start, end);
        }
        else
        {
            start = index + 1;
            index = Partition(input, n, start, end);
        }
    }

    for(int i = 0; i < k; ++i)
        output[i] = input[i];
}
```

采用这种思路是有限制的。我们需要修改输入的数组，因为函数 Partition 会调整数组中数字的顺序。如果面试官要求不能修改输入的数组，那么我们该怎么办呢？

❖ 解法二：时间复杂度为 $O(n \log k)$ 的算法，特别适合处理海量数据

我们可以先创建一个大小为 k 的数据容器来存储最小的 k 个数字，接下来每次从输入的 n 个整数中读入一个数。如果容器中已有的数字少于 k 个，则直接把这次读入的整数放入容器之中；如果容器中已有 k 个数字了，也

就是容器已满，此时我们不能再插入新的数字而只能替换已有的数字。找出这已有的 k 个数中的最大值，然后拿这次待插入的整数和最大值进行比较。如果待插入的值比当前已有的最大值小，则用这个数替换当前已有的最大值；如果待插入的值比当前已有的最大值还要大，那么这个数不可能是最小的 k 个整数之一，于是我们可以抛弃这个整数。

因此，当容器满了之后，我们要做 3 件事情：一是在 k 个整数中找到最大数；二是有可能在这个容器中删除最大数；三是有可能要插入一个新的数字。如果用一棵二叉树来实现这个数据容器，那么我们能在 $O(\log k)$ 时间内实现这 3 步操作。因此，对于 n 个输入数字而言，总的时间效率就是 $O(n \log k)$ 。

我们可以选择用不同的二叉树来实现这个数据容器。由于每次都需要找到 k 个整数中的最大数字，我们很容易想到用最大堆。在最大堆中，根节点的值总是大于它的子树中任意节点的值。于是我们每次可以在 $O(1)$ 时间内得到已有的 k 个数字中的最大值，但需要 $O(\log k)$ 时间完成删除及插入操作。

我们自己从头实现一个最大堆需要一定的代码，这在面试短短的几十分钟内很难完成。我们还可以采用红黑树来实现我们的容器。红黑树通过把节点分为红、黑两种颜色并根据一些规则确保树在一定程度上是平衡的，从而保证在红黑树中的查找、删除和插入操作都只需要 $O(\log k)$ 时间。在 STL 中，`set` 和 `multiset` 都是基于红黑树实现的。如果面试官不反对我们用 STL 中的数据容器，那么我们可以直接拿过来用。下面是基于 STL 中的 `multiset` 的参考代码：

```
typedef multiset<int, greater<int>> intSet;
typedef multiset<int, greater<int>>::iterator setIterator;

void GetLeastNumbers(const vector<int>& data, intSet& leastNumbers, int k)
{
    leastNumbers.clear();

    if(k < 1 || data.size() < k)
        return;

    vector<int>::const_iterator iter = data.begin();
    for(; iter != data.end(); ++ iter)
    {
        if((leastNumbers.size()) < k)
            leastNumbers.insert(*iter);

        else
```

```

    {
        setIterator iterGreatest = leastNumbers.begin();

        if(*iter < *(leastNumbers.begin()))
        {
            leastNumbers.erase(iterGreatest);
            leastNumbers.insert(*iter);
        }
    }
}

```

❖ 解法比较

基于函数 Partition 的第一种解法的平均时间复杂度是 $O(n)$ ，比第二种解法要快，但同时它也有明显的限制，比如会修改输入的数组。

第二种解法虽然慢一点，但它有两个明显的优点。一是没有修改输入的数据（代码中的变量 data）。我们每次只是从 data 中读入数字，所有的写操作都是在容器 leastNumbers 中进行的。二是该算法适合海量数据的输入（包括百度在内的多家公司非常喜欢与海量输入数据相关的问题）。假设题目是要求从海量的数据中找出最小的 k 个数字，由于内存的大小是有限的，有可能不能把这些海量的数据一次性全部载入内存。这个时候，我们可以从辅助存储空间（如硬盘）中每次读入一个数字，根据 GetLeastNumbers 的方式判断是不是需要放入容器 leastNumbers 即可。这种思路只要求内存能够容纳 leastNumbers 即可，因此它最适合的情形就是 n 很大并且 k 较小的问题。

我们可以用表 5.1 总结这两种解法的特点。

表 5.1 两种解法的特点比较

	基于 Partition 函数的解法	基于堆或者红黑树的解法
时间复杂度	$O(n)$	$O(n \log k)$
是否需要修改输入数组	是	否
是否适用于海量数据	否	是

由于这两种解法各有优缺点，各自适用于不同的场合，因此应聘者在动手做题之前要先问清楚题目的要求，包括输入的数据量有多大、能否一次性载入内存、是否允许交换输入数据中数字的顺序等。



面试小提示:

如果面试时遇到的面试题有多种解法，并且每种解法都各有优缺点，那么我们要向面试官问清楚题目的要求、输入的特点，从而选择最合适的方法。



源代码:

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/40_KLeastNumbers



测试用例:

- 功能测试（输入的数组中有相同的数字；输入的数组中没有相同的数字）。
- 边界值测试（输入的 k 等于 1 或者等于数组的长度）。
- 特殊输入测试（ k 小于 1； k 大于数组的长度；指向数组的指针为 NULL）。



本题考点:

- 考查应聘者对时间复杂度的分析能力。面试的时候每想出一种解法，我们都要能分析出这种解法的时间复杂度是多少。
- 如果采用第一种思路，则本题考查应聘者对 Partition 函数的理解。这个函数既是快速排序的基础，也可以用来查找 n 个数中第 k 大的数字。
- 如果采用第二种思路，则本题考查应聘者对堆、红黑树等数据结构的理解。当需要在某个数据容器内频繁查找及替换最大值时，我们要想到二叉树是一个合适的选择，并能想到用堆或者红黑树等特殊的二叉树来实现。

面试题 41：数据流中的中位数

题目：如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

由于数据是从一个数据流中读出来的，因而数据的数目随着时间的变化而增加。如果用一个数据容器来保存从流中读出来的数据，则当有新的数据从流中读出来时，这些数据就插入数据容器。这个数据容器用什么数据结构定义最合适呢？

数组是最简单的数据容器。如果数组没有排序，则可以用 Partition 函数找出数组中的中位数（详见面试题 39）。在没有排序的数组中插入一个数字和找出中位数的时间复杂度分别是 $O(1)$ 和 $O(n)$ 。

我们还可以在往数组里插入新数据时让数组保持排序。这时由于可能需要移动 $O(n)$ 个数，因此需要 $O(n)$ 时间才能完成插入操作。在已经排好序的数组中找出中位数是一个简单的操作，只需要 $O(1)$ 时间即可完成。

排序的链表是另外一种选择。我们需要 $O(n)$ 时间才能在链表中找到合适的位置插入新的数据。如果定义两个指针指向链表中间的节点（如果链表的节点数目是奇数，那么这两个指针指向同一个节点），那么可以在 $O(1)$ 时间内得出中位数。此时的时间复杂度与基于排序的数组的时间复杂度一样。

二叉搜索树可以把插入新数据的平均时间降低到 $O(\log n)$ 。但是，当二叉搜索树极度不平衡从而看起来像一个排序的链表时，插入新数据的时间仍然是 $O(n)$ 。为了得到中位数，可以在二叉树节点中添加一个表示子树节点数目的字段。有了这个字段，可以在平均 $O(\log n)$ 时间内得到中位数，但最差情况仍然需要 $O(n)$ 时间。

为了避免二叉搜索树的最差情况，还可以利用平衡的二叉搜索树，即 AVL 树。通常 AVL 树的平衡因子是左、右子树的高度差。可以稍作修改，把 AVL 的平衡因子改为左、右子树节点数目之差。有了这个改动，可以用 $O(\log n)$ 时间往 AVL 树中添加一个新节点，同时用 $O(1)$ 时间得到所有节点的中位数。

AVL 树的时间效率很高，但大部分编程语言的函数库中都没有实现这个数据结构。应聘者在短短几十分钟内实现 AVL 树的插入操作是非常困难

的，于是我们不得不再分析还有没有其他的方法。

如图 5.1 所示，如果数据在容器中已经排序，那么中位数可以由 P_1 和 P_2 指向的数得到。如果容器中数据的数目是奇数，那么 P_1 和 P_2 指向同一个数据。

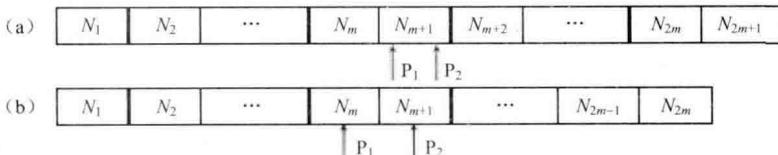


图 5.1 容器中数据被中间的一个或两个数据分隔成两部分：(a) 数据的数目是奇数；
(b) 数据的数目是偶数

我们注意到整个数据容器被分隔成两部分。位于容器左边部分的数据比右边的数据小。另外， P_1 指向的数据是左边部分最大的数， P_2 指向的数据是左边部分最小的数。

如果能够保证数据容器左边的数据都小于右边的数据，那么即使左、右两边内部的数据没有排序，也可以根据左边最大的数及右边最小的数得到中位数。如何快速从一个数据容器中找出最大数？用最大堆实现这个数据容器，因为位于堆顶的就是最大的数据。同样，也可以快速从最小堆中找出最小数。

因此，可以用如下思路来解决这个问题：用一个最大堆实现左边的数据容器，用一个最小堆实现右边的数据容器。往堆中插入一个数据的时间效率是 $O(\log n)$ 。由于只需要 $O(1)$ 时间就可以得到位于堆顶的数据，因此得到中位数的时间复杂度是 $O(1)$ 。

表 5.2 总结了使用没有排序的数组、排序的数组、排序的链表、二叉搜索树、AVL 树、最大堆和最小堆几种不同的数据结构的时间复杂度。

表 5.2 使用没有排序的数组、排序的数组、排序的链表、二叉搜索树、AVL 树、最大堆和最小堆等不同数据结构的时间复杂度

数据结构	插入的时间复杂度	得到中位数的时间复杂度
没有排序的数组	$O(1)$	$O(n)$
排序的数组	$O(n)$	$O(1)$
排序的链表	$O(n)$	$O(1)$

续表

数据结构	插入的时间复杂度	得到中位数的时间复杂度
二叉搜索树	平均 $O(\log n)$, 最差 $O(n)$	平均 $O(\log n)$, 最差 $O(n)$
AVL 树	$O(\log n)$	$O(1)$
最大堆和最小堆	$O(\log n)$	$O(1)$

接下来考虑用最大堆和最小堆实现的一些细节。首先要保证数据平均分配到两个堆中，因此两个堆中数据的数目之差不能超过 1。为了实现平均分配，可以在数据的总数目是偶数时把新数据插入最小堆，否则插入最大堆。

还要保证最大堆中的所有数据都要小于最小堆中的数据。当数据的总数目是偶数时，按照前面的分配规则会把新的数据插入最小堆。如果此时这个新的数据比最大堆中的一些数据要小，那该怎么办呢？

可以先把这个新的数据插入最大堆，接着把最大堆中最大的数字拿出来插入最小堆。由于最终插入最小堆的数字是原最大堆中最大的数字，这样就保证了最小堆中所有数字都大于最大堆中的数字。

当需要把一个数据插入最大堆，但这个数据小于最小堆里的一些数据时，这个情形和前面类似，请大家自己分析。

以下是用 C++ 实现的参考代码。我们基于 STL 中的函数 `push_heap`、`pop_heap` 及 `vector` 实现堆。比较仿函数 `less` 和 `greater` 分别用来实现最大堆和最小堆。

```
template<typename T> class DynamicArray
{
public:
    void Insert(T num)
    {
        if(((min.size() + max.size()) & 1) == 0)
        {
            if(max.size() > 0 && num < max[0])
            {
                max.push_back(num);
                push_heap(max.begin(), max.end(), less<T>());
                num = max[0];
                pop_heap(max.begin(), max.end(), less<T>());
                max.pop_back();
            }
            min.push_back(num);
        }
    }
}
```

```

        push_heap(min.begin(), min.end(), greater<T>());
    }
    else
    {
        if(min.size() > 0 && min[0] < num)
        {
            min.push_back(num);
            push_heap(min.begin(), min.end(), greater<T>());
            num = min[0];
            pop_heap(min.begin(), min.end(), greater<T>());
            min.pop_back();
        }

        max.push_back(num);
        push_heap(max.begin(), max.end(), less<T>());
    }
}

T GetMedian()
{
    int size = min.size() + max.size();
    if(size == 0)
        throw exception("No numbers are available");

    T median = 0;
    if((size & 1) == 1)
        median = min[0];
    else
        median = (min[0] + max[0]) / 2;

    return median;
}

private:
    vector<T> min;
    vector<T> max;
};
```

在上述代码中，`min` 是一个最小堆，`max` 是一个最大堆。函数 `Insert` 用来插入从数据流中读出来的数据，函数 `GetMedian` 用来得到已有所有数据的中位数。



源代码：

本题完整的源代码：

https://github.com/zhedahht/CodingInterviewChinese2/tree/master/41_StreamMedian