六星教育 WEB 前端面试圣经

常见排序算法的时间复杂度,空间复杂度

排序算法可以分为内部排序和外部排序，内部排序是数据记录在内存中进行排序，而外部排序是因排序的数据很大，一次不能容纳全部的排序记录，在排序过程中需要访问外存。常见的内部排序算法有：插入排序、希尔排序、选择排序、冒泡排序、归并排序、快速排序、堆排序、基数排序等。用一张图概括：

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

js 实现一个函数，完成超过范围的两个大整数相加功能

主要思路是通过将数字转换为字符串，然后每个字符串在按位相加。

```
function bigNumberAdd(number1, number2) {
let result = "", // 保存最后结果
carry = false; // 保留进位结果
// 将字符串转换为数组
number1 = number1.split("");
number2 = number2.split("");
// 当数组的长度都变为 0，并且最终不再进位时，结束循环
while (number1.length || number2.length || carry) {
// 每次将最后的数字进行相加，使用~~的好处是，即使返回值为 undefined 也能转换为 0
carry += ~~number1.pop() + ~~number2.pop();
// 取加法结果的个位加入最终结果
result = carry % 10 + result;
```

```
        // 判断是否需要进位，true 和 false 的值在加法中会被转换为 1 和 0
        carry = carry > 9;
        }
        // 返回最终结果
        return result;
        }
```

js 如何实现数组扁平化？

```
        // 这一种方法通过递归来实现，当元素为数组时递归调用，兼容性好
        function flattenArray(array) {
        if (!Array.isArray(array)) return;
        let result = [];
        result = array.reduce(function (pre, item) {
        // 判断元素是否为数组，如果为数组则递归调用，如果不是则加入结果数组中
        return pre.concat(Array.isArray(item) ? flattenArray(item) : item);
        }, []);
        return result;
        }
        // 这一种方法是利用了 toString 方法，它的一个缺点是改变了元素的类型，只适合于数组中元素都是整数的情况
        function flattenArray(array) {
        return array.toString().split(",").map(function (item) {
        return +item;
        })
```

```
}
```

js 如何实现数组去重？

```
function unique(array) {
if (!Array.isArray(array) || array.length <= 1) return;
var result = [];
array.forEach(function (item) {
if (result.indexOf(item) === -1) {
result.push(item);
}
})
return result;
}
function unique(array) {
if (!Array.isArray(array) || array.length <= 1) return;
return [...new Set(array)];
}
```

如何求数组的最大值和最小值？

```
var arr = [6, 4, 1, 8, 2, 11, 23];
console.log(Math.max.apply(null, arr))
```

如何求两个数的最大公约数？

基本思想是采用辗转相除的方法，用大的数去除以小的那个数，然后再用小的数去除以的得到的余数，一直这样递归下去，直到余数为 0 时，最后的被除数就是两个数的最大公约数。

```
function getMaxCommonDivisor(a, b) {
  if (b === 0) return a;
  return getMaxCommonDivisor(b, a % b);
}
```

如何求两个数的最小公倍数？

```
基本思想是采用将两个数相乘，然后除以它们的最大公约数
function getMinCommonMultiple(a, b){
return a * b / getMaxCommonDivisor(a, b);
}
```

实现 IndexOf 方法？

```
function indexFun(array, val) {
if (!Array.isArray(array)) return;
let length = array.length;
for (let i = 0; i < length; i++) {
if (array[i] === val) {
return i;
}
}
return -1;
}
```

判断一个字符串是否为回文字符串？

```
function isPalindrome(str) {
```

```
let reg = /[\W_]/g, // 匹配所有非单词的字符以及下划线
newStr = str.replace(reg, "").toLowerCase(), // 替换为空字符并将大写字母转换为小写
reverseStr = newStr.split("").reverse().join(""); // 将字符串反转
return reverseStr === newStr;
}
```

实现一个累加函数的功能比如 sum(1,2,3)(2).valueOf()

```
function sum(...args) {
let result = 0;
result = args.reduce(function (pre, item) {
return pre + item;
}, 0);
let add = function (...args) {
result = args.reduce(function (pre, item) {
return pre + item;
}, result);
return add;
};
add.valueOf = function () {
console.log(result);
}
return add;
}
```

使用 reduce 方法实现 forEach、map、filter

```
// forEach
function forEachUseReduce(array, handler) {
array.reduce(function (pre, item, index) {
handler(item, index);
});
}
// map
function mapUseReduce(array, handler) {
let result = [];
array.reduce(function (pre, item, index) {
let mapItem = handler(item, index);
result.push(mapItem);
});
return result;
}
// filter
function filterUseReduce(array, handler) {
let result = [];
array.reduce(function (pre, item, index) {
if (handler(item, index)) {
result.push(item);
```

```
        }
        });
        return result;
    }
```

设计一个简单的任务队列，要求分别在 1,3,4 秒后打印出 "1", "2", "3"

```
        class Queue {
        constructor() {
        this.queue = [];
        this.time = 0;
        }
        addTask(task, t) {
        this.time += t;
        this.queue.push([task, this.time]);
        return this;
        }
        start() {
        this.queue.forEach(item => {
        setTimeout(() => {
        item[0]();
        }, item[1]);
        })
        }
```

```
}
```

如何查找一篇英文文章中出现频率最高的单词？

```
function findMostWord(article) {
// 合法性判断
if (!article) return;
// 参数处理
article = article.trim().toLowerCase();
let wordList = article.match(/[a-z]+/g),
visited = [],
maxNum = 0,
maxWord = "";
article = " " + wordList.join(" ") + " ";
// 遍历判断单词出现次数
wordList.forEach(function (item) {
if (visited.indexOf(item) < 0) {
let word = new RegExp(" " + item + " ", "g"),
num = article.match(word).length;

if (num > maxNum) {
maxNum = num;
maxWord = item;
}
```

```
        }
        });
        return maxWord + " " + maxNum;
    }
```

JavaScript 明星问题

题目：

有 n 个人，其中一个明星和 n-1 个群众，群众都认识明星，明星不认识任何群众，群众和群众之间的认识关系不知道，现有一个

函数 foo(A, B)，若 A 认识 B 返回 true，若 A 不认识 B 返回 false，试设计一种算法找出明星，并给出时间复杂度。

思路：

（1）第一种方法我们可以直接使用双层循环遍历的方式，每一个人都和其他人进行判断，如果一个人谁都不认识，那么他就是明星。

这一种方法的时间复杂度为 O(n^2)。

（2）上一种方法没有充分利用题目所给的条件，其实我们每一次比较，都可以排除一个人的可能。比如如果 A 认识 B，那么说明

A 就不会是明星，因此 A 就可以从数组中移除。如果 A 不认识 B，那么说明 B 不可能是明星，因此 B 就可以从数组中移

除。因此每一次判断都能够减少一个可能性，我们只需要从数组从前往后进行遍历，每次移除一个不可能的人，直到数组中只剩

一人为止，那么这个人就是明星。这一种方法的时间复杂度为 O(n)。

正 负 数 组 求 和

题目：有两个数组，一个数组里存放的是正整数，另一个数组里存放的是负整数，都是无序的，现在从两个数组里各拿一个，使得它们的和 最接近零。

思路：（1）首先我们可以对两个数组分别进行排序，正数数组按从小到大排序，负数数组按从大到小排序。排序完成后我们使用两个指针分 别指向两个数组的首部，判断两个指针的和。如果和大于 0，则负数指针往后移动一个位置，如果和小于 0，则正数指针往后移动 一个位置，每一次记录和的值，和当前保存下来的最小值进行比较。

JavaScript 反转单向链表

需要将一个单向链表反转。思路很简单，使用三个变量分别表示当前节点和当前节点的前后节点，虽然这题很简单，但是却是 一道面试常考题。

思路是从头节点往后遍历，先获取下一个节点，然后将当前节点的 next 设置为前一个节点，然后再继续循环。

```
var reverseList = function(head) {
    // 判断下变量边界问题
    if (!head || !head.next) return head;
    // 初始设置为空，因为第一个节点反转后就是尾部，尾部节点指向 null
    let pre = null;
    let current = head;
    let next;
    // 判断当前节点是否为空
    // 不为空就先获取当前节点的下一节点
    // 然后把当前节点的 next 设为上一个节点
    // 然后把 current 设为下一个节点，pre 设为当前节点
    while(current) {
        next = current.next;
        current.next = pre;
        pre = current;
        current = next;
    }
```

```
    return pre;
};
```

JavaScript 二叉树相关性质

节点的度：一个节点含有的子树的个数称为该节点的度；

叶节点或终端节点：度为零的节点；

节点的层次：从根开始定义起，根为第 1 层，根的子节点为第 2 层，以此类推。

树的高度或深度：树中节点的最大层次。

在非空二叉树中，第 i 层的结点总数不超过 2^(i-1)，i>=1。

深度为 h 的二叉树最多有 2^h-1 个结点(h>=1)，最少有 h 个结点。

对于任意一棵二叉树，如果其叶结点数为 N0，而度数为 2 的结点总数为 N2，则 N0 = N2+1；

给定 N 个节点，能构成 h(N) 种不同的二叉树。h(N)为卡特兰数的第 N 项。(2n)!/(n!(n+1)!)。

二叉树的前序遍历，首先访问根结点，然后遍历左子树，最后遍历右子树。简记根-左-右。

二叉树的中序遍历，首先遍历左子树，然后访问根结点，最后遍历右子树。简记左-根-右。

二叉树的后序遍历，首先遍历左子树，然后遍历右子树，最后访问根结点。简记左-右-根。

二叉树是非线性数据结构，但是顺序存储结构和链式存储结构都能存储。

一个带权的无向连通图的最小生成树的权值之和是唯一的。

只有一个结点的二叉树的度为 0 。

二叉树的度是以节点的最大的度数定义的。

树的后序遍历序列等同于该树对应的二叉树的中序序列。

树的先序遍历序列等同于该树对应的二叉树的先序序列。

线索二叉树的线索实际上指向的是相应遍历序列特定结点的前驱结点和后继结点，所以先写出二叉树的中序遍历序列：debxac，中序遍历中在 x 左边和右边的字符，就是它在中序线索化的左、右线索，即 b、a 。

递归式的先序遍历一个 n 节点，深度为 d 的二叉树，需要栈空间的大小为 O（d），因为二叉树并不一定是平衡的， 也就是深度 d！=logn，有可能 d>>logn。所以栈大小应该是 O（d）

一棵具有 N 个结点的二叉树的前序序列和后序序列正好相反 ,则该二叉树一定满足该二叉树只有左子树或只有右子树，即该二叉树一定是一条链（二叉树的高度为 N，高度等于结点数）。

引入二叉线索树的目的是加快查找结点的前驱或后继的速度。

二叉树线索化后，先序线索化与后序线索化最多有 1 个空指针域，而中序线索化最多有 2 个空指针域。

不管是几叉树，节点数等于=分叉数+1

任何一棵二叉树的叶子结点在先序、中序和后序遍历中的相对次序不发生改变。

JavaScript 满二叉树

对于一棵二叉树 ,如果每一个非叶子节点都存在左右子树 ,并且二叉树中所有的叶子节点都在同一层中 ,这样的二叉树称为满 二叉树。

JavaScript 完全二叉树

对于一棵具有 n 个节点的二叉树按照层次编号 ,同时 ,左右子树按照先左后右编号 ,如果编号为 i 的节点与同样深度的满 二叉树中编号为 i 的节点在满二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。

性质：

具有 n 个结点的完全二叉树的深度为 K =[log2n」+1(取下整数)

有 N 个结点的完全二叉树各结点如果用顺序方式存储，则结点之间有如下关系： 若 I 为结点编号（从 1 开始编号）则 如果 I>1，则其父结点的编号为 I/2

完全二叉树，如果 2 * I <= N，则其左儿子（即左子树的根结点）的编号为 2 * I；若 2 * I > N，则无左儿子；如 果 2 * I + 1 <= N，则其右儿子的结点编号为 2 * I + 1；若 2 * I + 1 > N，则无右儿子。

JavaScript 平衡二叉查找树（AVL）

平衡二叉查找树具有如下几个性质：

可以是空树。

假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过 1。

平衡二叉树是为了解决二叉查找树中出现链式结构（只有左子树或只有右子树）的情况，这样的情况出现后对我们的查找没有 一点帮帮助，反而增加了维护的成本。

平衡因子使用两个字母来表示。第一个字母表示最小不平衡子树根结点的平衡因子，第二个字母表示最小不平衡子树较高子树 的根结点的平衡因子。根据不同的情况使用不同的方法来调整失衡的子树。

JavaScript B-树

B-树主要用于文件系统以及部分数据库索引，如 MongoDB。使用 B-树来作为数据库的索引主要是为了减少查找是磁盘的 I/O 次数。试想，如果我们使用二叉查找树来作为索引，那么查找次数的最坏情况等于二叉查找树的高度，由于索引存储在磁盘中， 我们每次都只能加载对应索引的磁盘页进入内存中比较，那么磁盘的 I/O 次数就等于索引树的高度。所以采用一种办法来减少 索引树的高度是提高索引效率的关键。

B-树是一种多路平衡查找树，它的每一个节点最多包含 K 个子节点，K 被称为 B-树的阶，K 的大小取决于磁盘页的大小。每 个节点中的元素从小到大排列，节点当中 k-1 个元素正好是 k 个孩子包含的元素的值域分划。简单来说就是以前一个磁盘页 只存储一个索引的值，但 B-树中一个磁盘页中存储了多个索引的值，因此在相同的比较范围内，B-树相对于一般的二叉查找树 的高度更小。其实它的主要目的就是每次尽可能多的将索引值加载入内存中进行比较 ,以此来减少磁盘的 I/O 次数，其实就查 找次数而言，和二叉查找树比较差不了多少，只是说这个比较过程是在内存中完成的，速度更快而已。

JavaScript B+树

B+ 树相对于 B-树有着更好的查找性能，根据 B-树我们可以知道，要想加快索引速度的方法就是尽量减少磁盘 I/O 的次数。B+ 树相对于 B-的主要变化是，每个中间节点中不再包含卫星数据，只有叶子节点包含卫星数据，每个父节点都出现在子节点中，叶子节点依次相连，形成一个顺序链表。中间节点不包含卫星数据，只用来作为索引使用，这意味着每一个磁盘页中能够 包含更多的索引值。因此 B+ 树的高度相对于 B-来说更低，所以磁盘的 I/O 次数更少。由于叶子节点依次相连，并且包含 了父节点，所以可以通过叶子节点来找到对应的值。同时 B+ 树所有查询都要查找到叶子节点，查询性能比 B-树稳定。

JavaScript 数据库索引

数据库以 B 树或者 B+ 树格式来储存的数据的，一张表是根据主键来构建的树的结构。因此如果想查找其他字段，就需要建 立索引，我对于索引的理解是它就是以某个字段为关键字的 B 树文件，通过这个 B 树文件就能够提高数据查找的效率。但是 由于我们需要维护的是平衡树的结构，因此对于数据的写入、修改、删除就会变慢，因为这有可能会涉及到树的平衡调整。

JavaScript 红黑树

红黑树是一种自平衡的二叉查找树，它主要是为了解决不平衡的二叉查找树的查找效率不高的缺点。红黑树保证了从根到叶子 节点的最长路径不会超过最短路径的两倍。

红黑树的有具体的规则：

1.节点是红色或黑色。

2.根节点是黑色。

3.每个叶子节点都是黑色的空节点（NIL 节点）。

4 每个红色节点的两个子节点都是黑色。(从每个叶子到根的所有路径上不能有两个连续的红色节点)

5.从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点。

当红黑树发生删除和插入导致红黑树不满足这些规则时，需要通过处理，使其重新满足这些规则。

JavaScript Huffman 树

给定 n 权值作为 n 个叶子节点，构造一棵二叉树，若这棵二叉树的带权路径长度达到最小，则称这样的二叉树为最优二叉树，也称为 Huffman 树。

利用 Huffman 树对每一个字符编码 ,该编码又称为 Huffman 编码 ,Huffman 编码是一种前缀编码 ,即一个字符的编码 不是另一个字符编码的前缀。

性质：

1.

对应一组权重构造出来的 Huffman 树一般不是唯一的

2.

3.

Huffman 树具有最小的带权路径长度

4.

5.

Huffman 树中没有度为 1 的结点

6.

7.

哈夫曼树是带权路径长度最短的树，路径上权值较大的结点离根较近

8.

9.

Huffman 树的带权路径长度 WPL 等于各叶子结点的带权路径长度之和

10.

JavaScript 二叉查找树

二叉查找树是一种特殊的二叉树，相对较小的值保存在左节点中，较大的值保存在右节点中，这一特性使得查找的效率很高， 对于数值型和非数值型数据，比如字母和字符串，都是如此。

实现树节点类：

```
// 节点类，树的节点
class Node {
  constructor(value) {
    this.value = value;
    this.left = null;
    this.right = null;
  }

  show() {
    console.log(this.value);
  }
}
```

实现二叉查找树类：

```
class BinarySearchTree {
```

```
    constructor() {
      this.root = null
    }

}
```

实现树的节点插入方法

节点插入的基本思想是将插入节点和当前节点做比较，如果比当前节点值小，并且没有左子树，那么将节点作为左叶子节点，否则继续和左子树进行比较。如果比当前节点值大，并且没有右子树，则将节点作为右叶子节点，否则继续和右子树进行比较。 循环这个过程直到找到合适的插入位置。

```
    insert(value) {

      let newNode = new Node(value);

      // 判断根节点是否为空，如果不为空则递归插入到树中
      if (this.root === null) {
        this.root = newNode;
      } else {
        this.insertNode(this.root, newNode);
      }
    }

```

```
    insertNode(node, newNode) {

    // 将插入节点的值与当前节点的进行比较，如果比当前节点小，则递归判断左子树，如果比当前节点大，则递归判断右子树。
    if (newNode.value < node.value) {
      if (node.left === null) {
        node.left = newNode;
      } else {
        this.insertNode(node.left, newNode);
      }
    } else {
      if (node.right === null) {
        node.right = newNode;
      } else {
        this.insertNode(node.right, newNode);
      }
    }

  }
```

通过递归实现树的先序、中序、后序遍历

```
  // 先序遍历通过递归实现
  // 先序遍历则先打印当前节点，再递归打印左子节点和右子节点。
  preOrderTraverse() {
```

```
    this.preOrderTraverseNode(this.root);
  }

  preOrderTraverseNode(node) {
    if (node !== null) {
      node.show();
      this.preOrderTraverseNode(node.left);
      this.preOrderTraverseNode(node.right);
    }
  }

  // 中序遍历通过递归实现
  // 中序遍历则先递归打印左子节点，再打印当前节点，最后再递归打印右子节点。
  inOrderTraverse() {
    this.inOrderTraverseNode(this.root);
  }

  inOrderTraverseNode(node) {
    if (node !== null) {
      this.inOrderTraverseNode(node.left);
      node.show();
      this.inOrderTraverseNode(node.right);
```

```
    }
  }

  // 后序遍历通过递归实现
  // 后序遍历则先递归打印左子节点和右子节点，最后再打印当前节点。
  postOrderTraverse() {
    this.postOrderTraverseNode(this.root);
  }

  postOrderTraverseNode(node) {
    if (node !== null) {
      this.postOrderTraverseNode(node.left);
      this.postOrderTraverseNode(node.right);
      node.show();
    }
  }
```

通过循环实现树的先序、中序、后序遍历

```
  // 先序遍历通过循环实现
  // 通过栈来实现循环先序遍历，首先将根节点入栈。然后进入循环，每次循环开始，当前节点出栈，打印当前节点，然后将
  // 右子节点入栈，再将左子节点入栈，然后进入下一循环，直到栈为空结束循环。
  preOrderTraverseByStack() {
    let stack = [];
```

```
    // 现将根节点入栈，开始遍历
    stack.push(this.root);

    while (stack.length > 0) {

        // 从栈中获取当前节点
        let node = stack.pop();

        // 执行节点操作
        node.show();

        // 判断节点是否还有左右子节点，如果存在则加入栈中，注意，由于中序遍历先序遍历是先访问根
        // 再访问左和右子节点，因此左右子节点的入栈顺序应该是反过来的
        if (node.right) {
            stack.push(node.right);
        }

        if (node.left) {
            stack.push(node.left);
        }
    }
```

```
    }

    // 中序遍历通过循环实现
    // 中序遍历先将所有的左子节点入栈，如果左子节点为 null 时，打印栈顶元素，然后判断该元素是否有右子树，如果有
    // 则将右子树作为起点重复上面的过程，一直循环直到栈为空并且节点为空时。
    inOrderTraverseByStack() {
        let stack = [],
            node = this.root;

        // 中序遍历是先左再根最后右
        // 所以首先应该先把最左边节点遍历到底依次 push 进栈
        // 当左边没有节点时，就打印栈顶元素，然后寻找右节点
        while (stack.length > 0 || node) {
            if (node) {
                stack.push(node);
                node = node.left;
            } else {
                node = stack.pop();
                node.show();
                node = node.right;
            }
        }
```

```
    }

    // 后序遍历通过循环来实现
    // 使用两个栈来是实现，先将根节点放入栈 1 中，然后进入循环，每次循环将栈顶元素加入栈 2，再依次将左节点和右节点依次
    // 加入栈 1 中，然后进入下一次循环，直到栈 1 的长度为 0。最后再循环打印栈 2 的值。
    postOrderTraverseByStack() {
      let stack1 = [],
        stack2 = [],
        node = null;

      // 后序遍历是先左再右最后根
      // 所以对于一个栈来说，应该先 push 根节点
      // 然后 push 右节点，最后 push 左节点

      stack1.push(this.root);

      while (stack1.length > 0) {
        node = stack1.pop();

        stack2.push(node);

        if (node.left) {
```

```
            stack1.push(node.left);
        }

        if (node.right) {
            stack1.push(node.right);
        }

    }

    while (stack2.length > 0) {
        node = stack2.pop();
        node.show();
    }
}
```

实现寻找最大最小节点值

```
// 寻找最小值，在最左边的叶子节点上
findMinNode(root) {
    let node = root;

    while (node && node.left) {
        node = node.left;
    }
```

```
    return node;
  }

  // 寻找最大值，在最右边的叶子节点上

  findMaxNode(root) {
    let node = root;

    while (node && node.right) {
      node = node.right;
    }

    return node;
  }
```

实现寻找特定大小节点值

```
  // 寻找特定值
  find(value) {
    return this.findNode(this.root, value);
  }

  findNode(node, value) {
```

```
    if (node === null) {
      return node;
    }
    if (value < node.value) {
      return this.findNode(node.left, value);
    } else if (value > node.value) {
      return this.findNode(node.right, value);
    } else {
      return node;
    }
  }
```

实现移除节点值

移除节点的基本思想是，首先找到需要移除的节点的位置，然后判断该节点是否有叶节点。如果没有叶节点，则直接删除，如 果有一个叶子节点，则用这个叶子节点替换当前的位置。如果有两个叶子节点，则去右子树中找到最小的节点来替换当前节点。

```
  // 移除指定值节点
  remove(value) {
    this.removeNode(this.root, value);
  }
  removeNode(node, value) {
```

```
        if (node === null) {
            return node;
        }

        // 寻找指定节点
        if (value < node.value) {
            node.left = this.removeNode(node.left, value);
            return node;
        } else if (value > node.value) {
            node.right = this.removeNode(node.right, value);
            return node;
        } else { // 找到节点

            // 第一种情况——没有叶节点
            if (node.left === null && node.right === null) {
                node = null;
                return node;
            }

            // 第二种情况——一个只有一个子节点的节点，将节点替换为节点的子节点
            if (node.left === null) {
                node = node.right;
```

```
            return node;
        } else if (node.right === null) {
            node = node.left;
        }

        // 第三种情况——一个有两个子节点的节点，去右子树中找到最小的节点，用它的值来替换当前节点
        // 的值，保持树的特性，然后将替换的节点去掉
        let aux = this.findMinNode(node.right);
        node.value = aux.value;
        node.right = this.removeNode(node.right, aux);
        return node;
    }
}
```