响应字段	作用
Location	客户端重定向到某个 URL
Proxy-Authenticate	向代理服务器发送验证信息
Server	服务器名字
WWW-Authenticate	获取资源需要的验证信息

实体字段	作用
Allow	资源的正确请求方式
Content-Encoding	内容的编码格式
Content-Language	内容使用的语言
Content-Length	request body 长度
Content-Location	返回数据的备用地址
Content-MD5	Base64 加密格式的内容 MD5 检验值
Content-Range	内容的位置范围
Content-Type	内容的媒体类型
Expires	内容的过期时间
Last_modified	内容的最后修改时间

4 DNS

> DNS 的作用就是通过域名查询到具体的 IP。

- 因为 IP 存在数字和英文的组合（IPv6），很不利于人类记忆，所以就出现了域名。你可以把域名看成是某个 IP 的别名， DNS 就是去查询这个别名的真正名称是什么

> 在 TCP 握手之前就已经进行了 DNS 查询， 这个查询是操作系统自己做的。
> 当你在浏览器中想访问 www.google.com 时，会进行一下操作

- 操作系统会首先在本地缓存中查询

- 没有的话会去系统配置的 DNS 服务器中查询
- 如果这时候还没得话，会直接去 DNS 根服务器查询， 这一步查询会找出负责 com 这个一级域名的服务器
- 然后去该服务器查询 google 这个二级域名
- 接下来三级域名的查询其实是我们配置的，你可以给 www 这个域名配置一个 IP，然后还可以给别的三级域名配置一个 IP

> 以上介绍的是 DNS 迭代查询， 还有种是递归查询， 区别就是前者是由客户端去做请求，后者是由系统配置的 DNS 服务器做请求，得到结果后将数据返回给客户端。

二、数据结构

2.1 栈

概念

- 栈是一个线性结构，在计算机中是一个相当常见的数据结构。
- 栈的特点是只能在某一端添加或删除数据， 遵循先进后出的原则

实现

> 每种数据结构都可以用很多种方式来实现， 其实可以把栈看成是数组的一个子集，所以这里使用数组来实现

```js
class Stack {
  constructor() {
    this.stack = []
  }
  push(item) {
    this.stack.push(item)
  }
  pop() {
    this.stack.pop()
  }
  peek() {
    return this.stack[this.getCount() - 1]
```

```
  }
  getCount() {
    return this.stack.length
  }
  isEmpty() {
    return this.getCount() === 0
  }
}
```

应用

> 匹配括号， 可以通过栈的特性来完成

```js
var isValid = function (s) {
  let map = {
    '(': -1,
    ')': 1,
    '[': -2,
    ']': 2,
    '{': -3,
    '}': 3
  }
  let stack = []
  for (let i = 0; i < s.length; i++) {
    if (map[s [i]] < 0) {
      stack.push(s [i])
    } else {
      let last = stack.pop()
      if (map[last] + map[s [i]] != 0) return false
    }
  }
  if (stack.length > 0) return false
  return true
};
```

2.2 队列

概念

> 队列一个线性结构，特点是在某一端添加数据，在另一端删除数据， 遵循先进先出的原则

实现

> 这里会讲解两种实现队列的方式，分别是单链队列和循环队列

- 单链队列

```js
class Queue {
  constructor() {
    this.queue = []
  }
  enQueue(item) {
    this.queue.push(item)
  }
  deQueue() {
    return this.queue.shift()
  }
  getHeader() {
    return this.queue[0]
  }
  getLength() {
    return this.queue.length
  }
  isEmpty() {
    return this.getLength() === 0
  }
}
```

> 因为单链队列在出队操作的时候需要 `O(n)` 的时间复杂度，所以引入了循环队列。循环队列的出队操作平均是 `O(1)` 的时间复杂度

- 循环队列

```js
class SqQueue {
  constructor(length) {
    this.queue = new Array(length + 1)
    // 队头
    this.first = 0
    // 队尾
    this.last = 0
```

```
    // 当前队列大小
    this.size = 0
  }
  enQueue(item) {
    // 判断队尾 + 1 是否为队头
    // 如果是就代表需要扩容数组
    // % this.queue.length 是为了防止数组越界
    if (this.first === (this.last + 1) % this.queue.length) {
      this.resize(this.getLength() * 2 + 1)
    }
    this.queue[this.last] = item
    this.size++
    this.last = (this.last + 1) % this.queue.length
  }
  deQueue() {
    if (this.isEmpty()) {
      throw Error( 'Queue is empty')
    }
    let r = this.queue[this.first]
    this.queue[this.first] = null
    this.first = (this.first + 1) % this.queue.length
    this.size--
    // 判断当前队列大小是否过小
    // 为了保证不浪费空间，在队列空间等于总长度四分之一时
    // 且不为 2 时缩小总长度为当前的一半
    if (this.size === this.getLength() / 4 && this.getLength() / 2 !== 0) {
      this.resize(this.getLength() / 2)
    }
    return r
  }
  getHeader() {
    if (this.isEmpty()) {
      throw Error( 'Queue is empty')
    }
    return this.queue[this.first]
  }
  getLength() {
    return this.queue.length - 1
  }
  isEmpty() {
    return this.first === this.last
  }
  resize(length) {
    let q = new Array(length)
    for (let i = 0; i < length; i++) {
      q [i] = this.queue[(i + this.first) % this.queue.length]
    }
```

```
    this.queue = q
    this.first = 0
    this.last = this.size
  }
}
```

2.3 链表

概念

> 链表是一个线性结构， 同时也是一个天然的递归结构。链表结构可以充分利用计算机内存空间， 实现灵活的内存动态管理。但是链表失去了数组随机读取的优点， 同时链表由于增加了结点的指针域， 空间开销比较大

实现

- 单向链表

```js
class Node {
  constructor(v, next) {
    this.value = v
    this.next = next
  }
}
class LinkList {
  constructor() {
    // 链表长度
    this.size = 0
    // 虚拟头部
    this.dummyNode = new Node(null, null)
  }
  find(header, index, currentIndex) {
    if (index === currentIndex) return header
    return this.find(header.next, index, currentIndex + 1)
  }
  addNode(v, index) {
    this.checkIndex(index)
    // 当往链表末尾插入时， prev.next 为空
    // 其他情况时， 因为要插入节点，所以插入的节点
    // 的 next 应该是 prev.next
```

```
    // 然后设置 prev.next 为插入的节点
    let prev = this.find(this.dummyNode, index, 0)
    prev.next = new Node(v, prev.next)
    this.size++
    return prev.next
  }
  insertNode(v, index) {
    return this.addNode(v, index)
  }
  addToFirst(v) {
    return this.addNode(v, 0)
  }
  addToLast(v) {
    return this.addNode(v, this.size)
  }
  removeNode(index, isLast) {
    this.checkIndex(index)
    index = isLast ? index - 1 : index
    let prev = this.find(this.dummyNode, index, 0)
    let node = prev.next
    prev.next = node.next
    node.next = null
    this.size--
    return node
  }
  removeFirstNode() {
    return this.removeNode(0)
  }
  removeLastNode() {
    return this.removeNode(this.size, true)
  }
  checkIndex(index) {
    if (index < 0 || index > this.size) throw Error( 'Index error')
  }
  getNode(index) {
    this.checkIndex(index)
    if (this.isEmpty()) return
    return this.find(this.dummyNode, index, 0).next
  }
  isEmpty() {
    return this.size === 0
  }
  getSize() {
    return this.size
  }
}
```

2.4 树

二叉树

- 树拥有很多种结构， 二叉树是树中最常用的结构， 同时也是一个天然的递归结构。
- 二叉树拥有一个根节点，每个节点至多拥有两个子节点，分别为：左节点和右节点 。树的最底部节点称之为叶节点， 当一颗树的叶数量数量为满时，该树可以称之为满二叉树

二分搜索树

- 二分搜索树也是二叉树，拥有二叉树的特性 。但是区别在于二分搜索树每个节点的值都比他的左子树的值大， 比右子树的值小
- 这种存储方式很适合于数据搜索 。如下图所示， 当需要查找 6 的时候， 因为需要查找的值比根节点的值大，所以只需要在根节点的右子树上寻找，大大提高了搜索效率

- 实现

```
class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
  }
}
class BST {
  constructor() {
    this.root = null
    this.size = 0
  }
  getSize() {
    return this.size
  }
  isEmpty() {
    return this.size === 0
  }
  addNode(v) {
    this.root = this._addChild(this.root, v)
  }
```

```
    // 添加节点时，需要比较添加的节点值和当前
    // 节点值的大小
    _addChild(node, v) {
      if (!node) {
        this.size++
        return new Node(v)
      }
      if (node.value > v) {
        node.left = this._addChild(node.left, v)
      } else if (node.value < v) {
        node.right = this._addChild(node.right, v)
      }
      return node
    }
  }
```

- 以上是最基本的二分搜索树实现，接下来实现树的遍历。

> 对于树的遍历来说，有三种遍历方法，分别是先序遍历 、中序遍历 、后序遍历 。三种遍历的区别在于何时访问节点 。在遍历树的过程中，每个节点都会遍历三次，分别是遍历到自己， 遍历左子树和遍历右子树 。如果需要实现先序遍历，那么只需要第一次遍历到节点时进行操作即可

```js
// 先序遍历可用于打印树的结构
// 先序遍历先访问根节点，然后访问左节点， 最后访问右节点。
preTraversal() {
  this._pre(this.root)
}
_pre(node) {
  if (node) {
    console.log(node.value)
    this._pre(node.left)
    this._pre(node.right)
  }
}
// 中序遍历可用于排序
// 对于 BST 来说， 中序遍历可以实现一次遍历就
// 得到有序的值
// 中序遍历表示先访问左节点，然后访问根节点， 最后访问右节点。
midTraversal() {
  this._mid(this.root)
}
_mid(node) {
```

```
    if (node) {
      this._mid(node.left)
      console.log(node.value)
      this._mid(node.right)
    }
  }
  // 后序遍历可用于先操作子节点
  // 再操作父节点的场景
  // 后序遍历表示先访问左节点，然后访问右节点， 最后访问根节点。
  backTraversal() {
    this._back(this.root)
  }
  _back(node) {
    if (node) {
      this._back(node.left)
      this._back(node.right)
      console.log(node.value)
    }
  }
```

> 以上的这几种遍历都可以称之为深度遍历，对应的还有种遍历叫做广度遍历，也就是一层层地遍历树 。对于广度遍历来说， 我们需要利用之前讲过的队列结构来完成

```js
breadthTraversal()  {
  if ( !this.root) return null
  let q = new Queue()
  // 将根节点入队
  q.enQueue(this.root)
  // 循环判断队列是否为空，为空
  // 代表树遍历完毕
  while ( !q.isEmpty()) {
    // 将队首出队，判断是否有左右子树
    // 有的话，就先左后右入队
    let n = q.deQueue()
    console.log(n.value)
    if (n.left) q.enQueue(n.left)
    if (n.right) q.enQueue(n.right)
  }
}
```

> 接下来先介绍如何在树中寻找最小值或最大数 。因为二分搜索树的特性，所以最小值一定在根节点的最左边， 最大值相反

```js
getMin() {
  return this._getMin(this.root).value
}
_getMin(node) {
  if ( !node.left) return node
  return this._getMin(node.left)
}
getMax() {
  return this._getMax(this.root).value
}
_getMax(node) {
  if ( !node.right) return node
  return this._getMin(node.right)
}
```

> 向上取整和向下取整， 这两个操作是相反的，所以代码也是类似的， 这里只介绍如何向下取整 。既然是向下取整，那么根据二分搜索树的特性，值一定在根节点的左侧 。只需要一直遍历左子树直到当前节点的值不再大于等于需要的值，然后判断节点是否还拥有右子树 。如果有的话， 继续上面的递归判断

```js
floor(v) {
  let node = this._floor(this.root, v)
  return node ? node.value : null
}
_floor(node, v) {
  if ( !node) return null
  if (node.value === v) return v
  // 如果当前节点值还比需要的值大，就继续递归
  if (node.value > v) {
    return this._floor(node.left, v)
  }
  // 判断当前节点是否拥有右子树
  let right = this._floor(node.right, v)
  if (right) return right
  return node
}
```

> 排名， 这是用于获取给定值的排名或者排名第几的节点的值， 这两个操作也是相反的，所以这个只介绍如何获取排名第几的节点的值 。对于这个操作而言，我们需要略微的改造点代码，让每个节点拥有一个 size 属性 。该属性表示该节点下有多少子节点（包含自身)

```js
class Node {
  constructor(value) {
    this.value = value
    this.left = null
    this.right = null
    // 修改代码
    this.size = 1
  }
}
// 新增代码
_getSize(node) {
  return node ? node.size : 0
}
_addChild(node, v) {
  if (!node) {
    return new Node(v)
  }
  if (node.value > v) {
    // 修改代码
    node.size++
    node.left = this._addChild(node.left, v)
  } else if (node.value < v) {
    // 修改代码
    node.size++
    node.right = this._addChild(node.right, v)
  }
  return node
}
select(k) {
  let node = this._select(this.root, k)
  return node ? node.value : null
}
_select(node, k) {
  if (!node) return null
  // 先获取左子树下有几个节点
  let size = node.left ? node.left.size : 0
  // 判断 size 是否大于 k
  // 如果大于 k，代表所需要的节点在左节点
  if (size > k) return this._select(node.left, k)
  // 如果小于 k，代表所需要的节点在右节点
```

```
    // 注意这里需要重新计算 k，减去根节点除了右子树的节点数量
    if (size < k) return this._select(node.right, k - size - 1)
    return node
  }
```

> 接下来讲解的是二分搜索树中最难实现的部分：删除节点 。因为对于删除节点来说，会存在以下几种情况

- 需要删除的节点没有子树
- 需要删除的节点只有一条子树
- 需要删除的节点有左右两条树
- 对于前两种情况很好解决，但是第三种情况就有难度了，所以先来实现相对简单的操作：删除最小节点，对于删除最小节点来说， 是不存在第三种情况的，删除最大节点操作是和删除最小节点相反的，所以这里也就不再赘述

```js
  delectMin() {
    this.root = this._delectMin(this.root)
    console.log(this.root)
  }
  _delectMin(node) {
    // 一直递归左子树
    // 如果左子树为空，就判断节点是否拥有右子树
    // 有右子树的话就把需要删除的节点替换为右子树
    if ((node != null) & !node.left) return node.right
    node.left = this._delectMin(node.left)
    // 最后需要重新维护下节点的 `size`
    node.size = this._getSize(node.left) + this._getSize(node.right) + 1
    return node
  }
```

- 最后讲解的就是如何删除任意节点了 。对于这个操作， `T.Hibbard` 在 `1962` 年提出了解决这个难题的办法，也就是如何解决第三种情况。
- 当遇到这种情况时， 需要取出当前节点的后继节点 (也就是当前节点右子树的最小节点) 来替换需要删除的节点 。然后将需要删除节点的左子树赋值给后继结点，右子树删除后继结点后赋值给他。
- 你如果对于这个解决办法有疑问的话， 可以这样考虑 。因为二分搜索树的特性， 父节点一定比所有左子节点大， 比所有右子节点小 。那么当需要删除父节点时， 势必需要拿出一个比父节点大的节点来替换父节点 。这个节点肯定不存在于左子树，必然存在于右子树 。然后又需要保持父节点都是比右子节点小的，那么就可以取出右子树中最小的那个节点来替换父节点

```js
  delect(v) {
    this.root = this._delect(this.root, v)
  }
  _delect(node, v) {
    if ( !node) return null
    // 寻找的节点比当前节点小 ，去左子树找
    if (node.value < v) {
      node.right = this._delect(node.right, v)
    } else if (node.value > v) {
      // 寻找的节点比当前节点大，去右子树找
      node.left = this._delect(node.left, v)
    } else {
      // 进入这个条件说明已经找到节点
      // 先判断节点是否拥有拥有左右子树中的一个
      // 是的话，将子树返回出去， 这里和 `_delectMin` 的操作一样
      if ( !node.left) return node.right
      if ( !node.right) return node.left
      // 进入这里，代表节点拥有左右子树
      // 先取出当前节点的后继结点，也就是取当前节点右子树的最小值
      let min = this._getMin(node.right)
      // 取出最小值后，删除最小值
      // 然后把删除节点后的子树赋值给最小值节点
      min.right = this._delectMin(node.right)
      // 左子树不动
      min.left = node.left
      node = min
    }
    // 维护 size
    node.size = this._getSize(node.left) + this._getSize(node.right) + 1
    return node
  }
```

2.5 堆

概念

- 堆通常是一个可以被看做一棵树的数组对象。
- 堆的实现通过构造二叉堆， 实为二叉树的一种 。这种数据结构具有以下性质。
- 任意节点小于（或大于）它的所有子节点 堆总是一棵完全树 。即除了最底层， 其他层的节点都被元素填满，且最底层从左到右填入 。
- 将根节点最大的堆叫做最大堆或大根堆，根节点最小的堆叫做最小堆或小根堆。
- 优先队列也完全可以用堆来实现， 操作是一模一样的。

实现大根堆

> 堆的每个节点的左边子节点索引是 `i * 2 + 1` ，右边是 `i * 2 + 2` ， 父节点是 `(i - 1) /2` 。

- 堆有两个核心的操作，分别是 `shiftUp` 和 `shiftDown` 。前者用于添加元素，后者用于删除根节点。
- `shiftUp` 的核心思路是一路将节点与父节点对比大小， 如果比父节点大，就和父节点交换位置。
- `shiftDown` 的核心思路是先将根节点和末尾交换位置，然后移除末尾元素 。接下来循环判断父节点和两个子节点的大小， 如果子节点大，就把最大的子节点和父节点交换

```js
class MaxHeap {
  constructor() {
    this.heap = []
  }
  size() {
    return this.heap.length
  }
  empty() {
    return this.size() == 0
  }
  add(item) {
    this.heap.push(item)
    this._shiftUp(this.size() - 1)
  }
  removeMax() {
    this._shiftDown(0)
  }
  getParentIndex(k) {
    return parseInt((k - 1) / 2)
  }
  getLeftIndex(k) {
    return k * 2 + 1
  }
  _shiftUp(k) {
    // 如果当前节点比父节点大，就交换
    while (this.heap[k] > this.heap[this.getParentIndex(k)]) {
      this._swap(k, this.getParentIndex(k))
      // 将索引变成父节点
      k = this.getParentIndex(k)
```

```
        }
      }
      _shiftDown(k) {
        // 交换首位并删除末尾
        this._swap(k, this.size() - 1)
        this.heap.splice(this.size() - 1, 1)
        // 判断节点是否有左孩子， 因为二叉堆的特性，有右必有左
        while (this.getLeftIndex(k) < this.size()) {
          let j = this.getLeftIndex(k)
          // 判断是否有右孩子，并且右孩子是否大于左孩子
          if (j + 1 < this.size() && this.heap[j + 1] > this.heap[j]) j++
          // 判断父节点是否已经比子节点都大
          if (this.heap[k] >= this.heap[j]) break
          this._swap(k, j)
          k = j
        }
      }
      _swap(left, right) {
        let rightValue = this.heap[right]
        this.heap[right] = this.heap[left]
        this.heap[left] = rightValue
      }
    }
```

三、算法

3.1 时间复杂度

- 通常使用最差的时间复杂度来衡量一个算法的好坏。
- 常数时间 `O(1)` 代表这个操作和数据量没关系， 是一个固定时间的操作， 比如说四则运算。
- 对于一个算法来说， 可能会计算出如下操作次数 `aN + 1`， `N` 代表数据量 。那么该算法的时间复杂度就是 `O(N)` 。因为我们在计算时间复杂度的时候，数据量通常是非常大的， 这时候低阶项和常数项可以忽略不计。
- 当然可能会出现两个算法都是 `O(N)` 的时间复杂度，那么对比两个算法的好坏就要通过对比低阶项和常数项了

3.2 位运算

- 位运算在算法中很有用， 速度可以比四则运算快很多。
- 在学习位运算之前应该知道十进制如何转二进制， 二进制如何转十进制 。这里说明下简单的计算方式

- 十进制 `33` 可以看成是 `32 + 1` ， 并且 `33` 应该是六位二进制的（因为 `33` 近似 `32` ， 而 `32` 是 `2` 的五次方，所以是六位)， 那么 十进制 `33` 就是 `100001` ， 只要是 2 的次方，那么就是 `1` 否则都为 `0` 那么二进制 `100001` 同理， 首位是 `2^5` ，末位是 `2^0` ，相加得出 `33`

左移 <<

```
10 << 1 // -> 20
```

> 左移就是将二进制全部往左移动， `10` 在二进制中表示为 `1010` ， 左移一位后变成 `10100` ，转换为十进制也就是 `20` ，所以基本可以把左移看成以下公式 `a * (2 ^ b)`

算数右移 >>

```
10 >> 1 // -> 5
```

- 算数右移就是将二进制全部往右移动并去除多余的右边， 10 在二进制中表示为 `1010` ，右移一位后变成 `101` ，转换为十进制也就是 `5` ，所以基本可以把右移看成以下公式 `int v = a / (2 ^ b)`
- 右移很好用， 比如可以用在二分算法中取中间值

```
13 >> 1 // -> 6
```

按位操作

- 按位与

> 每一位都为 1， 结果才为 1

```
8 & 7 // -> 0
// 1000 & 0111 -> 0000 -> 0
```

- 按位或

> 其中一位为 1， 结果就是 1

```
8 | 7 // -> 15
// 1000 | 0111 -> 1111 -> 15
```

- 按位异或

> 每一位都不同， 结果才为 1

```
8 ^ 7 // -> 15
8 ^ 8 // -> 0
// 1000 ^ 0111 -> 1111 -> 15
// 1000 ^ 1000 -> 0000 -> 0
```

面试题：两个数不使用四则运算得出和

> 这道题中可以按位异或， 因为按位异或就是不进位加法， `8 ^ 8 = 0` 如果进位了，就是 `16` 了，所以我们只需要将两个数进行异或操作，然后进位 。那么也就是说两个二进制都是 1 的位置， 左边应该有一个进位 `1` ，所以可以得出以下公式 `a + b = (a ^ b) + ((a & b) << 1)` ，然后通过迭代的方式模拟加法

```js
function sum(a, b) {
    if (a == 0) return b
    if (b == 0) return a
    let newA = a ^ b
    let newB = (a & b) << 1
    return sum(newA, newB)
}
```

3.3 排序

冒泡排序

> 冒泡排序的原理如下， 从第一个元素开始， 把当前元素和下一个索引元素进行比较 。如果当前元素大，那么就交换位置， 重复操作直到比较到最后一个元素，那么此时最后一个元素就是该数组中最大的数 。下一轮重复以上操作，但是此时最后一个元素已经是最大数了，所以不需要再比较最后一个元素， 只需要比较到 `length - 1` 的位置

> 以下是实现该算法的代码

```js
function bubble(array) {
  checkArray(array);
  for (let i = array.length - 1; i > 0; i--) {
    // 从 0 到 `length - 1` 遍历
    for (let j = 0; j < i; j++) {
      if (array[j] > array[j + 1]) swap(array, j, j + 1)
    }
  }
  return array;
}
```

> 该算法的操作次数是一个等差数列 `n + (n - 1) + (n - 2) + 1` ， 去掉常数项以后得出时间复杂度是 `O(n * n)`

插入排序

> 入排序的原理如下 。第一个元素默认是已排序元素， 取出下一个元素和当前元素比较， 如果当前元素大就交换位置 。那么此时第一个元素就是当前的最小数，所以下次取出操作从第三个元素开始， 向前对比， 重复之前的操作

以下是实现该算法的代码

```js
function insertion(array) {
  checkArray(array);
  for (let i = 1; i < array.length; i++) {
```

```js
      for (let j = i - 1; j >= 0 && array[j] > array[j + 1]; j--)
        swap(array, j, j + 1);
    }
    return array;
  }
```

> 该算法的操作次数是一个等差数列 `n + (n - 1) + (n - 2) + 1` ， 去掉常数项以后得出时间复杂度是 `O(n * n)`

选择排序

> 选择排序的原理如下。遍历数组，设置最小值的索引为 0， 如果取出的值比当前最小值小，就替换最小值索引， 遍历完成后，将第一个元素和最小值索引上的值交换。如上操作后， 第一个元素就是数组中的最小值，下次遍历就可以从索引 1 开始重复上述操作

以下是实现该算法的代码

```js
function selection(array) {
  checkArray(array);
  for (let i = 0; i < array.length - 1; i++) {
    let minIndex = i;
    for (let j = i + 1; j < array.length; j++) {
      minIndex = array[j] < array[minIndex] ? j : minIndex;
    }
    swap(array, i, minIndex);
  }
  return array;
}
```

> 该算法的操作次数是一个等差数列 `n + (n - 1) + (n - 2) + 1` ， 去掉常数项以后得出时间复杂度是 `O(n * n)`

归并排序

> 归并排序的原理如下。递归的将数组两两分开直到最多包含两个元素，然后将数组排序合并，最终合并为排序好的数组。假设我有一组数组 `[3, 1, 2, 8, 9, 7, 6]`，中间数索引是 3，先排序数组 `[3, 1, 2, 8]`。在这个左边数组上，继续拆分直到变成数组包含两个元素（如果数组长度是奇数的话，会有一个拆分数组只包含一个元素）。然后排序数组 `[3, 1]` 和 `[2, 8]`，然后再排序数组 `[1, 3, 2, 8]`，这样左边数组就排序完成，然后按照以上思路排序右边数组，最后将数组 `[1, 2, 3, 8]` 和 `[6, 7, 9]` 排序

以下是实现该算法的代码

```js
function sort(array) {
  checkArray(array);
  mergeSort(array, 0, array.length - 1);
  return array;
}

function mergeSort(array, left, right) {
  // 左右索引相同说明已经只有一个数
  if (left === right) return;
  // 等同于 `left + (right - left) / 2`
  // 相比 `(left + right) / 2` 来说更加安全，不会溢出
  // 使用位运算是因为位运算比四则运算快
  let mid = parseInt(left + ((right - left) >> 1));
  mergeSort(array, left, mid);
  mergeSort(array, mid + 1, right);

  let help = [];
  let i = 0;
  let p1 = left;
  let p2 = mid + 1;
  while (p1 <= mid && p2 <= right) {
    help[i++] = array[p1] < array[p2] ? array[p1++] : array[p2++];
  }
  while (p1 <= mid) {
    help[i++] = array[p1++];
  }
  while (p2 <= right) {
    help[i++] = array[p2++];
  }
  for (let i = 0; i < help.length; i++) {
    array[left + i] = help[i];
```

```
  }
  return array;
}
```

> 以上算法使用了递归的思想。递归的本质就是压栈，每递归执行一次函数，就将该函数的信息（比如参数， 内部的变量，执行到的行数）压栈， 直到遇到终止条件，然后出栈并继续执行函数。对于以上递归函数的调用轨迹如下

```js
mergeSort(data, 0, 6) // mid = 3
 mergeSort(data, 0, 3) // mid = 1
   mergeSort(data, 0, 1) // mid = 0
     mergeSort(data, 0, 0) // 遇到终止, 回退到上一步
   mergeSort(data, 1, 1) // 遇到终止, 回退到上一步
   // 排序 p1 = 0, p2 = mid + 1 = 1
   // 回退到 `mergeSort(data, 0, 3)` 执行下一个递归
 mergeSort(2, 3) // mid = 2
   mergeSort(3, 3) // 遇到终止, 回退到上一步
 // 排序 p1 = 2, p2 = mid + 1 = 3
 // 回退到 `mergeSort(data, 0, 3)` 执行合并逻辑
 // 排序 p1 = 0, p2 = mid + 1 = 2
 // 执行完毕回退
 // 左边数组排序完毕，右边也是如上轨迹
```

> 该算法的操作次数是可以这样计算：递归了两次，每次数据量是数组的一半，并且最后把整个数组迭代了一次，所以得出表达式 `2T(N / 2) + T(N)`（`T` 代表时间， `N` 代表数据量）。根据该表达式可以套用 该公式 得出时间复杂度为 `O(N * logN)`

快排

> 快排的原理如下。随机选取一个数组中的值作为基准值，从左至右取值与基准值对比大小。比基准值小的放数组左边，大的放右边，对比完成后将基准值和第一个比基准值大的值交换位置。然后将数组以基准值的位置分为两部分， 继续递归以上操作。

以下是实现该算法的代码

```js
function sort(array) {
  checkArray(array);
  quickSort(array, 0, array.length - 1);
  return array;
}

function quickSort(array, left, right) {
  if (left < right) {
    swap(array, , right)
    // 随机取值，然后和末尾交换， 这样做比固定取一个位置的复杂度略低
    let indexs = part(array, parseInt(Math.random() * (right - left + 1)) +
    quickSort(array, left, indexs [0]);
    quickSort(array, indexs [1] + 1, right);
  }
}
function part(array, left, right) {
  let less = left - 1;
  let more = right;
  while (left < more) {
    if (array[left] < array[right]) {
      // 当前值比基准值小， `less` 和 `left` 都加一
       ++less;
       ++left;
    } else if (array[left] > array[right]) {
      // 当前值比基准值大，将当前值和右边的值交换
      // 并且不改变 `left`， 因为当前换过来的值还没有判断过大小
      swap(array, --more, left);
    } else {
      // 和基准值相同， 只移动下标
      left++;
    }
  }
  // 将基准值和比基准值大的第一个值交换位置
  // 这样数组就变成 `[比基准值小 ， 基准值 ， 比基准值大]`
  swap(array, right, more);
  return [less, more];
}
```

> 该算法的复杂度和归并排序是相同的，但是额外空间复杂度比归并排序少， 只需 `O(logN)` ， 并且相比归并排序来说，所需的常数时间也更少

面试题

> Sort Colors：该题目来自 LeetCode，题目需要我们将 `[2,0,2,1,1,0]` 排序成 `[0,0,1,1,2,2]` ， 这个问题就可以使用三路快排的思想

```js
var sortColors = function(nums) {
  let left = -1;
  let right = nums.length;
  let i = 0;
  // 下标如果遇到 right，说明已经排序完成
  while (i < right) {
    if (nums [i] == 0) {
      swap(nums, i++, ++left);
    } else if (nums [i] == 1) {
      i++;
    } else {
      swap(nums, i, --right);
    }
  }
};
```

3.4 链表

反转单向链表

> 该题目来自 LeetCode，题目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

```js
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
    };
```

3.5 树

二叉树的先序， 中序，后序遍历

- 先序遍历表示先访问根节点，然后访问左节点， 最后访问右节点。
- 中序遍历表示先访问左节点，然后访问根节点， 最后访问右节点。
- 后序遍历表示先访问左节点，然后访问右节点， 最后访问根节点

递归实现

递归实现相当简单，代码如下

```js
function TreeNode(val) {
  this.val = val;
  this.left = this.right = null;
}
var traversal = function(root) {
  if (root) {
    // 先序
    console.log(root);
    traversal(root.left);
    // 中序
    // console.log(root);
    traversal(root.right);
    // 后序
    // console.log(root);
  }
};
```

> 对于递归的实现来说， 只需要理解每个节点都会被访问三次就明白为什么这样实现了

非递归实现

> 非递归实现使用了栈的结构， 通过栈的先进后出模拟递归实现。

以下是先序遍历代码实现

```js
function pre(root) {
  if (root) {
    let stack = [];
    // 先将根节点 push
    stack.push(root);
    // 判断栈中是否为空
    while (stack.length > 0) {
      // 弹出栈顶元素
      root = stack.pop();
      console.log(root);
      // 因为先序遍历是先左后右，栈是先进后出结构
      // 所以先 push 右边再 push 左边
      if (root.right) {
        stack.push(root.right);
      }
      if (root.left) {
        stack.push(root.left);
      }
    }
  }
}
```

以下是中序遍历代码实现

```js
function mid(root) {
  if (root) {
    let stack = [];
    // 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
    // 对于最左边的叶节点来说，可以把它看成是两个 null 节点的父节点
    // 左边打印不出东西就把父节点拿出来打印，然后再看右节点
    while (stack.length > 0 || root) {
      if (root) {
        stack.push(root);
        root = root.left;
      } else {
        root = stack.pop();
        console.log(root);
        root = root.right;
      }
    }
```

```
  }
}
```

> 以下是后序遍历代码实现，该代码使用了两个栈来实现遍历，相比一个栈的遍历来说要容易理解很多

```js
function pos(root) {
  if (root) {
    let stack1 = [];
    let stack2 = [];
    // 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(root);
    while (stack1.length > 0) {
      root = stack1.pop();
      stack2.push(root);
      if (root.left) {
        stack1.push(root.left);
      }
      if (root.right) {
        stack1.push(root.right);
      }
    }
    while (stack2.length > 0) {
      console.log(s2.pop());
    }
  }
}
```

中序遍历的前驱后继节点

> 实现这个算法的前提是节点有一个 `parent` 的指针指向父节点，根节点指向 `null`

> 如图所示，该树的中序遍历结果是 `4, 2, 5, 1, 6, 3, 7`

前驱节点

> 对于节点 2 来说，他的前驱节点就是 4 ，按照中序遍历原则， 可以得出以下结论

- 如果选取的节点的左节点不为空，就找该左节点最右的节点 。对于节点 1 来说，他有左节点 2 ，那么节点 2 的最右节点就是 5
- 如果左节点为空，且目标节点是父节点的右节点，那么前驱节点为父节点 。对于节点 5 来说，没有左节点，且是节点 2 的右节点，所以节点 2 是前驱节点
- 如果左节点为空，且目标节点是父节点的左节点， 向上寻找到第一个是父节点的右节点的节点 。对于节点 6 来说，没有左节点，且是节点 3 的左节点，所以向上寻找到节点 1 ，发现节点 3 是节点 1 的右节点，所以节点 1 是节点 6 的前驱节点

以下是算法实现

```js
function predecessor(node) {
  if (!node) return
  // 结论 1
  if (node.left) {
    return getRight(node.left)
  } else {
    let parent = node.parent
    // 结论 2 3 的判断
    while(parent && parent.right === node) {
      node = parent
      parent = node.parent
    }
    return parent
  }
}
function getRight(node) {
  if (!node) return
  node = node.right
  while(node) node = node.right
  return node
}
```

后继节点

> 对于节点 2 来说，他的后继节点就是 5 ，按照中序遍历原则， 可以得出以下结论