1. 什么是数据结构和算法

解决问题方法的效率，跟数据的组织方式有关

什么是数据结构

> 数据结构就是在计算机中，存储和组织数据的方式

什么是算法

算法(Algorithm)的定义：

- 一个有限指令集，每条指令的描述不依赖于语言
- 接受一些输入（有些情况下不需要输入）
- 产生输出
- 一定在有限步骤后终止

算法的通俗理解：

- Algorithm这个单词本意就是解决问题的办法/步骤逻辑
- 数据结构的实现，离不开算法

2. 数组结构

- js的数组就是API的调用
    - 因此，这里不讲
- 补充：普通语言的数组封装（比如Java的ArrayList）
    - 常见语言的数组不能存放不同 的数据类型，因此所有在封装时通常存放在数组中的是Object类型
    - 常见语言的数组容量不会自动改变（需要进行扩容操作）
    - 常见语言的数组进行中间插入和删除操作性能比较低

3. 栈结构

3.1. 认识栈结构

数组

- 我们知道数组是一种线性结构，并且可以在数组的**任意位置**插入和删除数据
- 但是有时候，我们为了实现某些功能，必须对这种**任意性**加以限制
- 而 **栈**和 **队列**就是比较常见的 **受限的线性结构**

> 栈（stack），它是一种受限的线性表，后进先出（LIFO）

- 其限制是仅允许在表的一端进行插入和删除运算。这一端被称为**栈顶**，相对地，把另一端称为**栈底**
- LIFO(last in first out)表示就是后进入的元素，第一个弹出栈空间
- 向一个栈插入新元素又被称为进栈、入栈、或者压栈
- 从一个栈删除元素又被称作出栈或者退栈

编程中的栈实现-函数调用栈

A调用B，B调用C，C调用D
当前栈顺序：栈底->A->B->C->D->栈顶

3.2. 栈结构面试题

有六个元素6,5,4,3,2,1的顺序进栈，问下列哪一个不是合法的出栈序列：

- A: 5,4,3,6,1,2
- B: 4,5,3,2,1,6
- C: 3,4,6,5,2,1
- D: 2,3,4,1,5,6
  答案：C

3.3. 栈结构的实现

实现栈结构有两种比较常见的方式：

- 基于数组实现
- 基于链表实现

什么是链表?

- 也是一种数据结构，还没有学习，JavaScript中并没有自带链表结构
- 后续会自己来实现链表结构，并且对比链表和数组的区别

基于数组实现：

```
<script>
    // Method:一般和某一个对象实例有联系，写在对象内部

    // 封装栈类
    function Stack() {
        // 栈中的属性
        this.items = []

        // 栈的相关操作
        // 1. 将元素压入栈

        // (1) 给某一个对象的实例添加方法。每一个对象都拥有function 这样的方法
        // this.push=function(){}

        // (2)给整个类添加方法。原型上面的方法是共享的，更节省内容，效率更高
        Stack.prototype.push = function (element) {
            this.items.push(element)
        }

        // 2. 从栈中取出元素
        Stack.prototype.pop = function () {
            return this.items.pop()
        }

        // 3. 查看一下栈顶元素
        Stack.prototype.peek = function () {
            return this.items[this.items.length - 1]
        }
```

```
        // 4. 判断栈是否为空
        Stack.prototype.isEmpty = function () {
            return this.items.length == 0
        }

        // 5. 获取栈中元素的个数
        Stack.prototype.size = function () {
            return this.items.length
        }

        // 6. toString()方法
        Stack.prototype.toString = function () {
            // 20 10 12 8 7 一个元素，一个空格的方式显示出来
            var resultString = ''
            for (var i = 0; i < this.items.length; i++) {

                resultString += this.items[i] + ' '
            }
            return resultString
        }
    }


    // 栈的使用
    var s = new Stack()
    s.push(10)
    s.push(20)
    s.push(30)
    s.push(40)
    s.push(50)
    s.push(60)

    s.pop()
    s.pop()
    // alert(s)

    // alert(s.peek())

    // alert(s.isEmpty())

    // alert(s.toString())
</script>
```

3.4. 栈的操作

栈常见操作：

- push(element)：添加一个新元素到栈顶位置
- pop():移除栈顶的元素，同时返回被移除的元素
- peek():返回栈顶的元素，不对栈做任何修改（这个方法不会移除栈顶的元素，仅仅是返回它）
- isEmpty():如果栈中无元素就返回true,否则返回false
- size():返回栈里的元素个数，这个方法和数组的length属性很相似
- toString():将栈结构的内容以字符形式返回

3.5. 栈的应用-十进制转二进制

```
// 函数：将十进制转成二进制
function dec2bin(decNumber) {
    // 1. 定义栈对象
    var stack = new Stack()

    // 2. 循环操作
    while (decNumber > 0) {
        // 2.1 获取余数并且放入栈中
        stack.push(decNumber % 2)

        // 2.2 获取整除后的结果作为下一次运行的数字
        decNumber = Math.floor(decNumber / 2)
    }

    // 3. 从栈中取出零和一
    var binaryString = ''
    while (!stack.isEmpty()) {
        binaryString += stack.pop()
    }
    return binaryString
}


// 测试十进制转二进制的函数
// alert(dec2bin(100))
// alert(dec2bin(10))
alert(dec2bin(1000))
```

4. 队列结构

4.1. 认识队列

> 队列(Queue)，它是一种受限的线性表，**先进先出**(FIFO,first in first out)

队列受限之处：

- 只允许在表的前端(front)进行删除操作
- 只允许在表的后端(rear)进行插入操作

生活中的队列结构：**排队**，优先排队的人优先处理

4.2. 队列的应用

- 打印队列
    - 有五份文档需要打印，这些文档会按照次序放入到打印队列中
    - 打印机会一次从队列中取出文档，优先放入的文档，优先被取出，并且对该文档进行打印
    - 以此类推，直到队列中不再有新的队列
- 线程队列
    - 开发中，为了让任务可以并行处理，通常会开启多个线程
    - 但是，不能让大量的线程同时运行处理任务
    - 这个时候如果有需要开启线程处理任务的情况，就会使用线程队列
    - 线程队列会按照次序来启动线程，并且处理对应的任务

4.3. 队列的实现

队列的实现和栈一样，有两种方案：

- 基于数组实现
- 基于链表实现

队列基于数组实现

```
<script>
    //封装队列类
    function Queue() {
        // 属性
        this.items = []


        // 方法

        // 1. 将元素加入到队列中
        Queue.prototype.enqueue = function (element) {
            this.items.push(element)
        }

        // 2. 从队列中删除前端元素
        Queue.prototype.dequeue = function () {
            return this.items.shift()
        }

        // 3. 查看前端的元素
        Queue.prototype.front = function () {
            return this.items[0]
        }

        // 4. 查看队列是否为空
        Queue.prototype.isEmpty = function () {
            return this.items.length == 0
        }

        // 5. 查看队列中元素的个数
        Queue.prototype.size = function () {
            return this.items.length
        }

        // 6. toString()方法
        Queue.prototype.toString = function () {
            var resultString = ''
            for (var i = 0; i < this.items.length; i++) {

                resultString += this.items[i] + ' '
            }
            return resultString
        }

    }


    // 使用队列
    var queue = new Queue()
```

```
    // 将元素加入到队列中
    queue.enqueue('abc')
    queue.enqueue('a')
    queue.enqueue('bc')
    queue.enqueue('ac')
    queue.enqueue('ab')

    // alert(queue)

    // 从队列中删除元素
    queue.dequeue()

    // alert(queue)

    // front方法
    // alert(queue.front())

    // isEmpty
    // alert(queue.isEmpty())

    // size()
    // alert(queue.size())

    // toString()
    // alert(queue.toString())
</script>
```

4.4. 队列的常见操作

- enqueue(element):向队列尾部添加一个（或多个）新的项
- dequeue():移除队列的第一（即排在队列最前面的）项，并返回被移除的元素
- front():返回队列中的第一个元素--最先被添加，也将是最先被移除的元素。队列不做任何变动（不移除元素，只返回元素信息--与Stack类的peek方法非常类似）
- isEmpty():如果队列中不包含任何元素，返回true，否则返回false
- size():返回队列包含的元素个数，与数组的length属性类似
- toString():将队列中的内容，转成字符串形式

4.5. 队列面试题-击鼓传花

游戏规则：
几个朋友围城一圈，开始数数，数到某个数字的人自动淘汰，最后剩下的这个人会获得胜利，请问最后剩下的是原来在哪一个位置上的人

封装一个基于队列的函数

- 参数：所有参赛人的姓名，基于的数字
- 结果：最终剩下的一人的姓名

```
// 面试题：击鼓传花
function passGame(nameList, num) {

    // 1. 创建一个队列结构
    var queue = new Queue()

    // 2. 将所有人依次加入到队列中
```

```
    for (var i = 0; i < nameList.length; i++) {
        queue.enqueue(nameList[i])
    }


    // 3. 开始数数字
    // 不是num的时候，重新加入到队列的末尾
    // 是num这个数字的时候，将其从队列中删除
    while (queue.size() > 1) {
        for (var i = 0; i < num - 1; i++) {
            // 3.1 num数字之前的人重新放入到队列的末尾
            queue.enqueue(queue.dequeue())
        }

        // 3.2 num对应的这个人直接淘汰掉，直接从队列中删除掉
        queue.dequeue()
    }

    // 4. 获取剩下的那个人
    alert(queue.size())
    var endName = queue.front()
    alert('最终剩下的人: ' + endName)

    return nameList.indexOf(endName)
}


// 测试击鼓传花
names=['a','b','c','d','e']
alert(passGame(names,3))
```

4.6. 优先级队列

优先级队列的特点：

- 普通的队列插入一个元素，数据会被放在后端，而且需要前面的所有元素都处理完成之后才会处理前面的数据
- 但是优先级队列，在插入一个元素的时候会考虑该数据的优先级
- 和其他优先级进行比较
- 比较完成后可以得出这个元素在队列中的正确位置
- 其他的处理方式和基本队列的处理方式一样

优先级队列的主要考虑的问题：

- 每个元素不再只是一个数据，而且包含数据的优先级
- 在添加方式中，根据优先级放入正确的位置

生活中优先级队列的应用：

- 机场登机的顺序，头等舱和商务舱优于经济舱
- 医院急诊科候诊室，优先处理病情比较严重的患者，一般按照排号顺序

4.7. 优先级队列的实现

```
<script>
    // 封装优先级队列
    function PriorityQueue() {
```

```
// 在PriorityQueue重新创建了一个类：可以理解成内部类

function QueueElement(element, priority) {
    this.element = element
    this.priority = priority
}

// 封装属性
this.items = []

// 1. 实现插入方法
PriorityQueue.prototype.enqueue = function (element, priority) {
    // 1. 创建QueueElement对象
    var queueElement = new QueueElement(element, priority)

    // 2. 判断队列是否为空
    if (this.items.length == 0) {
        this.items.push(queueElement)
    } else {
        var added=false
        for (var i = 0; i < this.items.length; i++) {
            if (queueElement.priority < this.items[i].priority) {
                this.items.splice(i, 0, queueElement)
                added = true
                break
            }
        }
        if (!added) {
            this.items.push(queueElement)
        }
    }
}

// 2. 从队列中删除前端元素
PriorityQueue.prototype.dequeue = function () {
    return this.items.shift()
}

// 3. 查看前端的元素
PriorityQueue.prototype.front = function () {
    return this.items[0]
}

// 4. 查看队列是否为空
PriorityQueue.prototype.isEmpty = function () {
    return this.items.length == 0
}

// 5. 查看队列中元素的个数
PriorityQueue.prototype.size = function () {
    return this.items.length
}

// 6. toString()方法
PriorityQueue.prototype.toString = function () {
    var resultString = ''
    for (var i = 0; i < this.items.length; i++) {
```

```
                resultString += this.items[i].element + '-' +
this.items[i].priority + ' '
            }
            return resultString
        }
    }

    // 测试代码
    var pq = new PriorityQueue()

    pq.enqueue('a', 1)
    pq.enqueue('b', 2)
    pq.enqueue('c', 3)
    pq.enqueue('z',1000)
    pq.enqueue('d', 4)
    pq.enqueue('e', 5)
    pq.enqueue('wang',0)
    pq.enqueue('sponge',-1)
    pq.enqueue('f', 6)
    pq.enqueue('g', 7)
    pq.enqueue('h', 8)

    alert(pq)
</script>
```