快速排序

```
（1）、确定基准数

我们把数组的第一个元素作为基准数。
2.将比这个数小全放左边，大于右边
3.在对左右区间重复第二步骤，直到区间只有一个数(条件就是数组中的元素要大于等于2个);返回排序好的左右数组
4.将有序的区间合并起来，这样整个数列就是有序的了；  合并好排序好的坐数组，基准数，排序好的右数组，并且返回
```

```
let arr = [31,23,34,2,13,234]
function quickSort(arr){
    let base_num = arr[0]
    let left_arr = []
    let right_arr = []
    //1
    for (var i = 1; i < arr.length; i++) {
        if(arr[i]<base_num){
            left_arr.push(arr[i])
        }else{
            right_arr.push(arr[i])
        }
    }
    //2
    if(left_arr.length>=2){
        left_arr=quickSort(left_arr)
    }
    if(right_arr.length>=2){
        right_arr=quickSort(right_arr)
    }
    //3.合并左数组,基准数，右数组
    return left_arr.concat(base_num,right_arr)
}
quickSort(arr)
```

二叉树

```
http://data.biancheng.net/view/192.html
简单地理解，满足以下两个条件的树就是二叉树：
本身是有序树；
树中包含的各个节点的度不能超过 2，即只能是 0、1 或者 2；
```

需求:输入某二叉树的前序遍历和中序遍历的结果，请重建该二叉树
例如：
输入前序遍历序列[1,2,4,7,3,5,6,8] 中序序列 [4,7,2,1,5,3,8,6] ,请重建二叉树并返回

两个栈实现一个队列

用两个栈来实现一个队列，完成队列的push和pop操作

```
js push()入栈，pop()出栈

let stack1=[]
let stack2=[]

//队列入操作
function push(node){
    stack1.push(node)
}
//队列出
function pop(){
    //1.把栈1--->栈2
    while(stack1.length){
        stack2.push(stack1.pop())
    }
    console.log("stack1",stack1,"stack2",stack2)
    //2.把栈2顶端的数据出栈
    let popVal = stack2.pop()
    //3.将栈2里面--->栈1，(还原数据)：
    while(stack2.length){
        stack1.push(stack2.pop())
    }
    return popVal
}
/*
1,2,3,4,5入队
出队1
入列6
出队2
出队3
*/
push(1)
push(2)
push(3)
push(4)
push(5)
console.log(stack1)
console.log(pop())
```

二叉树的创建,层次遍历

```
根据二叉树的层次遍历的序列结果，创建二叉树
例如：
层次遍历结果为：['a','b','c','d','#','#','e','#','f','#','#','#','#']
对应二叉树：# --->代表空节点
       a
     b     c
    d #   # e
  #   f     #  #
    #  #
```

层次遍历都是用队列来解决，注意空节点不会加到队列中

1.找到根节点
2.队列里面的a出队，把和a相关的b和c(把和出队元素相关的元素加入到队列),加到了队列

队列解决问题的算法模板：
```
while(队列不为空){
    1.将队列队首的元素出队(要么是整棵树的根节点，要么是子树的根节点)
    2.把和出队元素相关的元素加入到队列  (队列的左右孩子)
}
```

```
function treeNode(val){
    this.val=val
    this.left=null
    this.right=null
}
function createTree_levelOrder(level){
    let queue = []
    let root = null
    //
    if(levelOrderArr.length){
        let nodeVal = levelOrderArr.shift()
        root = new treeNode(nodeVal)
        queue.push(root)

        //2.循环将队列首的元素出队，把和出队元素相关的元素加入到队列
        //队列不为空
        while(queue.length){
            //1.将队列首 的元素出队
            let head = queue.shift()
            //2.把和出队元素相关的元素加入到队列(根节点的左右孩子)
            //a,创建左节点，将它加入到队列
            nodeVal = levelOrderArr.shift()
            if(nodeVal!=='#'){
                head.left=new treeNode(nodeVal)
                queue.push(head.left)
            }
            //b.创建右节点，将它加入到队列
            nodeVal=levelOrderArr.shift()
            if(nodeVal!=="#"){
                head.right=new treeNode(nodeVal)
                queue.push(head.right)
            }
        }
    }
    return root;
}
let levelOrderArr=['a','b','c','d','#','#','e','#','f','#','#','#','#']
createTree_levelOrder(levelOrderArr)
```

二叉树的创建，先序遍历

递归注意：
a.递归结束条件
b.递归的递推表达式（节点之间的关系）：根左右
c.递归的返回值，有的没有，本例返回创建好的树或则子树

栈的实现

```
/*
function Stack() {
    // 栈中的属性
    var items = []
    // 栈相关的方法
    // 压栈操作
    //写法1：每个对象都有这个方法
    this.push = function (element) {
        items.push(element)
    }

    // 出栈操作
    this.pop = function () {
        return items.pop()
    }
    // peek操作：查看栈顶
    this.peek = function () {
        return items[items.length - 1]
    }

    // 判断栈中的元素是否为空
    this.isEmpty = function () {
        return items.length == 0
    }
    // 获取栈中元素的个数
    this.size = function () {
        return items.length
    }
}
*/

function Stack() {
    var items = []
}
//写法2：基于原型，是共享的，更节省内存，推荐
Stack.prototype.push=function (element) {
    this.items.push(element)
}

stack.prototype.pop= function () {
    return this.items.pop()
}
// peek操作：查看栈顶
stack.prototype.peek= function () {
    return this.items[this.items.length - 1]
}
// 判断栈中的元素是否为空
stack.prototype.isEmpty= function () {
    return this.items.length == 0
```

```
}
// 获取栈中元素的个数
stack.prototype.size= function () {
    return this.items.length
}
stack.prototype.toString = function () {
    var result = "";
    for(var i = 0; i < this.items.length; i++){
      result += this.items[i] + ",";
    }
    return result;
}
var stack = new Stack()
stack.push(6)
stack.toString()
//    stack.push(5)
//    stack.pop()     // 5
//    stack.push(4)
//    stack.pop()     // 4
//    stack.push(3)
//    stack.pop()     // 3
//    stack.pop()     // 6
//    stack.push(2)
//    stack.push(1)
//    stack.pop()     // 1
//    stack.pop()     // 2
```

队列

```
function Queue() {
    this.items = []
};
//向队列插入元素
Queue.prototype.enqueue = function (element) {
    this.items.push(element)
}
//从队头删除元素
Queue.prototype.dequeue = function () {
    return this.items.shift();
}
//查看对头元素
Queue.prototype.front = function () {
    return this.items[0]
}
//查看队列的长度
Queue.prototype.size = function () {
    return this.items.length
}
//判断队列是否为空
Queue.prototype.isEmpty = function () {
    return items.length==0;
}
Queue.prototype.toString = function () {
        var result = "";
        console.log("this.items",this.items)
        for(var i = 0; i < this.items.length; i++){
          result += this.items[i] + ",";
```

```
        }
        return result;
}
// 创建队列对象
var queue = new Queue()

// 在队列中添加元素
queue.enqueue("abc")
queue.enqueue("cba")
queue.enqueue("nba")

// 查看一下队列前端元素
// alert(queue.front())
queue.toString()
```

队列击鼓传花面试题(循环队列)

围成一圈，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，请问剩下的是原来哪个位置的人

封装一个基于队列的函数，
    参数：所有参与者的名字，基于的数字
    结果：最终剩下的一个人的名字

```
function Queue() {
    this.items = []
};
//向队列插入元素
Queue.prototype.enqueue = function (element) {
    this.items.push(element)
}
//从队头删除元素
Queue.prototype.dequeue = function () {
    return this.items.shift();
}
//查看对头元素
Queue.prototype.front = function () {
    return this.items[0]
}
//查看队列的长度
Queue.prototype.size = function () {
    return this.items.length
}
//判断队列是否为空
Queue.prototype.isEmpty = function () {
    return items.length==0;
}
Queue.prototype.toString = function () {
        var result = "";
        console.log("this.items",this.items)
        for(var i = 0; i < this.items.length; i++){
          result += this.items[i] + " ";
        }
        return result;
}
```

```
// 击鼓传花规则：一群朋友围在一起做游戏，开始数数，数到某个数字的人自动淘汰，最后剩下的这个人胜利，借助输出这个人的索引值
function passGame(namelist, num) {
    // 1.啥都先别想，先创建一个队列结构
    var q = new Queue();

    // 2.将所有人都加入到队列结构中去
    for (var i = 0; i < namelist.length; i++) {
        q.enqueue(namelist[i])
    }

    // 3.所有人开始数数字，如果不是num的话移到队列末尾，如果是num的话从队列中删除
    while (q.size() > 1) {
        console.log("items",q.toString())
        for (var j = 0; j < num - 1; j++) {
            q.enqueue(q.dequeue())
        }
        //   4.num对应的这个人，把他从队列中移除出去
        q.dequeue()
    }
    // 5.获取最后剩下的那个人
    var endname = q.front();
    console.log("endname",endname)
    return namelist.indexOf(endname)
}
var names = ['John','Jack','Camila','Ingrid','Carl'];
var index = passGame(names, 7)
console.log("names",index)
```

优先级队列

```
function Queue() {
    this.items = []
    //定义内部类--封装插入元素
    function QueueElement(element, priority) {
        this.element = element;
        this.priority = priority;
    }
    Queue.prototype.enqueue = function(element, priority) {
        var qe = new QueueElement(element, priority);
        if(this.items.length === 0) {
            this.items.push(qe);
        } else {
            var added = false; //记录元素是否插入，没有就插入到items最后
            for(var i = 0; i < this.items.length; i++) {
                // 注意: 我们这里是数字越小，优先级越高
                if(priority < this.items[i].priority) {
                    this.items.splice(i, 0, qe); //splice:在数组指定位置插入,也可以删除指定元素["red", "blue", "grey"].splice(0,1)---->["blue", "grey"]
                    /*
                    slice:该方法并不会修改数组，而是返回一个子数组:
                    slice() 方法可从已有的数组中返回选定的元素。
arrayObject.slice(start,end)
                    */
                    added = true;
                    break;
                }
```

```
            }
            // 遍历完所有的元素，优先级都大于新插入的元素时，就插入到最后
            if(!added) {
                this.items.push(qe);
            }
        }
    }
};
//向队列插入元素
/*
Queue.prototype.enqueue = function (element) {
    this.items.push(element)
}
*/
//从队头删除元素
Queue.prototype.dequeue = function () {
    return this.items.shift();
}
//查看队头元素
Queue.prototype.front = function () {
    return this.items[0]
}
//查看队列的长度
Queue.prototype.size = function () {
    return this.items.length
}
//判断队列是否为空
Queue.prototype.isEmpty = function () {
    return items.length==0;
}
Queue.prototype.toString = function () {
        var result = "";
        console.log("this.items",this.items)
        for(var i = 0; i < this.items.length; i++){
          result += this.items[i] + " ";
        }
        return result;
}

var priorityQueue = new Queue();
priorityQueue.enqueue('abc', 10);
priorityQueue.enqueue('bc', 20);
priorityQueue.enqueue('cv', 6);
priorityQueue.enqueue('re', 34);
console.log("priorityQueue:",priorityQueue.toString());
console.log("priorityQueue:",priorityQueue);
```