# 数据结构

## 为什么需要学习数据结构?

### 1.语言是相通的?

人们常说，编程语言是相通的，掌握了一门，其他语言很容易就能掌握，个人认为这是一个似是而非的观点。

每门编程语言都离不开变量，数组，循环，条件判断这些概念，这似乎能够支持上面的观点，但是，每门编程语言都有自己的适用范围。都有自己擅长的事情，即便是有了node.js这种能够一统前后端的语言，也总有它不能胜任的工作，比如机器学习。像python这样的近乎万能的语言，也总有无能为力的时候，比如面对高性能计算，许多python库的底层实现可都是C语言哦。

多年的工作经验告诉我，真正相通的不是语言，而是数据结构和算法。

数据结构和算法是脱离编程语言而存在的，不同的语言有不同的实现版本，但内在的逻辑却不会有变化，所体现的编程思想不会有变化。

### 2. 一段亲身经历

我曾经在工作有过这样一次经历，我在后端通过websockt向前端发送数据，数据是一个具体的坐标，前端的同学得到坐标后，要在前端的中国地图上根据坐标显示一个光圈。这是一个非常简单的事情，但却遇到了麻烦，后端向前端推送数据是一个不定时的行为，有时1秒钟推了3条数据，有时3秒钟才推1条数据，当我推送数据频繁的时候，如果这些坐标都在地图上显示，地图会非常乱。长时间不推送数据时，前端页面不应该一直显示之前推送过的坐标，因为每一个坐标代表一个用户刚刚在我们的网站上做了一项操作。

于是，我们对前端显示做了限制，前端同一个时刻最多显示10个坐标，如果已经有10个，新来的坐标要把之前最早到来的坐标挤掉，每个坐标最多显示5秒钟。就是这样一个简单的要求，前端同学却迟迟不能实现该功能，因为他无法兼顾最多显示10个坐标和每个坐标最多显示5秒钟的要求。

后来，我让他用队列来实现。前端在收到坐标后，将坐标和收到时间构造成一个新的对象，一同放入到队列中，如果列队元素已经有10个，则把队列头部的元素删除，于此同时，每隔1秒钟就对队列里的元素检查一次，队列头部的元素都是最早到来的，如果当前时间距离到来时间超过5秒，则删除队列头部元素。

就这样，前端同学使用队列，非常快的实现了这个功能，而且对数据结构产生了浓厚的兴趣。

每当你怀疑学习数据结构的必要性和作用时，请提醒自己，如果你手里只有锤子，那么目光所及之处都是钉子。

### 3. 学习数据结构的目标

- 提高程序设计能力
- 提高算法能力
- 找工作面试的时候hold住面试官

数据结构的精髓在于总结提炼了许多存储管理和使用数据的模式，这些模式的背后是最精华的编程思想，这些思想的领悟需要时间，不要想当然的认为学会了几种数据结构就可以在工作中大显身手，但学会了数据结构，对自身能力的提升是不言而喻的。

当然，在没有参悟这些数据管理方式和编程思想之前，我们先学习具体的工具和方法。

### 4. 学习数据结构需要准备哪些知识

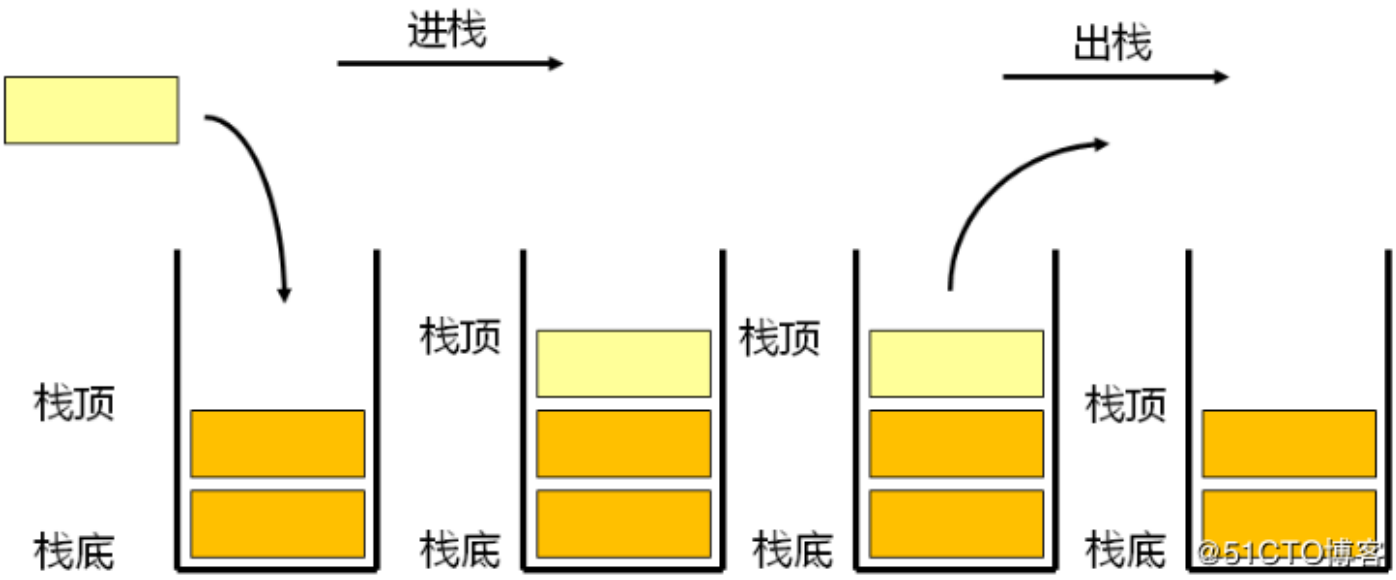
你需要熟练地使用数组这种数据类型，我想这没什么困难，再有就是知道如何在javascript中定义类，这里推荐一篇文章Javascript定义类（class）的三种方法 本系列课程主要使用构造函数法定义类。

## 数据结构之----栈

### 1、栈的定义

栈是一种特殊的线性表，仅能够在栈顶进行操作，有着先进后出(后进先出)的特性，下面这张图展示了栈的工作特点：

– 后进先出（Last In First Out）




对于栈的理解，你必须牢牢抓住一点，那便是你只能在栈顶进行操作，生活中有一个非常贴切的例子，玩羽毛球的同学都会买一桶羽毛球，羽毛球桶就是典型的栈结构。

每次取羽毛球时，都只能从顶部取，最底下的羽毛球，你是取不到的，用完了羽毛球后，也只能从顶部放回去。

## 2. 栈的实现

上一小节，我们对栈这种数据结构有了视觉上的接触和理解，接下来，我们要对栈进行定义，毕竟，我们写一个栈，为的是使用它，那么就必须先定义数据存储在哪里，提供什么样的方法。

### 2.1 数据存储

从数据存储的角度看，实现栈有两种方式，一种是以数组做基础，一种是以链表做基础，数组是最简单的实现方式，链表在后面会作为单独的一种数据结构来讲解。本次课程将使用数组来实现栈和队列，数组是大家平时用的最频繁的，也是最了解熟悉的数据类型。

我们先定义一个简单的Stack类

```
function Stack() {
    var items = [];  // 使用数组存储数据
};
```

数据将存储在items数组之中，现在，这个类没有任何方法。

## 2.2 栈的方法

栈有以下几个方法：

- push 添加一个元素到栈顶（向桶里放入一个羽毛球）
- pop 弹出栈顶元素（从桶里拿出一个羽毛球）
- top 返回栈顶元素，注意，不是弹出（看一眼桶里最顶端的羽毛球，但是不拿）
- isEmpty 判断栈是否为空（看看羽毛球是不是都用完了）
- size 返回栈里元素的个数（数一下桶里还有多少羽毛球）
- clear 清空栈（把桶里的羽毛球都倒出来扔掉）

下面，逐一实现这些方法

### 2.2.1 push方法

```
    // push方法向栈里压入一个元素
    this.push = function(item){
        items.push(item);
    };
```

### 2.2.2 pop方法

```
    // pop方法把栈顶的元素弹出
    this.pop = function(){
        return items.pop();
    };
```

数组的pop方法会删除最靠后的那个元素，同时return该元素

### 2.2.3 top方法

```
    // top 方法返回栈顶元素
    this.top = function(){
        return items[items.length - 1];
    };
```

top方法只是想查看一下最顶端的元素

### 2.2.4 isEmpty方法

```
    // isEmpty返回栈是否为空
    this.isEmpty = function(){
        return items.length == 0;
    };
```

### 2.2.5 size方法

```
    // size方法返回栈的大小
    this.size = function(){
        return items.length;
    };
```

### 2.2.6 clear方法

```
    // clear 清空栈
    this.clear = function(){
        items = []
    }
```

最终完成版的代码如下

```
function Stack() {
    var items = [];  // 使用数组存储数据

    // push方法向栈里压入一个元素
    this.push = function(item){
        items.push(item);
    };
```

```
    // pop方法把栈顶的元素弹出
    this.pop = function(){
        return items.pop();
    };

    // top 方法返回栈顶元素
    this.top = function(){
        return items[items.length - 1];
    };

    // isEmpty返回栈是否为空
    this.isEmpty = function(){
        return items.length == 0;
    };

    // size方法返回栈的大小
    this.size = function(){
        return items.length;
    };

    // clear 清空栈
    this.clear = function(){
        items = []
    }
}
```

### 2.3 被欺骗的错觉

看完上面的实现，难道你没有一种被欺骗的感觉么？传的那么神乎其神的数据结构，这里实现的栈，竟然就只是对数组做了一层封装而已啊！！！

只是做了一层封装么？请思考下面几个问题：

1. 给你一个数组，你可以通过索引操作任意一个元素，但是给你一个栈，你能操作任意元素么？栈提供的方法只允许你操作栈顶的元素，也就是数组的最后一个元素，这种限制其实提供给我们一种思考问题的方式，这个方式也就是栈的特性，后进先出。

2. 既然栈的底层实现其实就是数组，栈能做的事情，数组一样可以做啊，为什么弄出一个栈来，是不是多此一举？封装是为了隐藏实现细节，站在栈的肩膀上思考问题显然要比站在数组的肩膀上思考问题更方便，后面的练习题你将有所体会。

3. 既然栈的底层就是对数组的操作，而你平时对数组的使用已经到了非常熟练的程度了，那么请问问自己，为什么就从来都没有自己实现过一个栈呢？是你此前不知道栈的这个

概念，还是知道栈的概念但是不知道它有哪些具体方法？不论是哪一种情况，都表明栈对你来说是一个全新的知识，尽管底层的实现是那么的简单，可是越简单就越能说明问题，为啥，你自己就没想出栈这个东西？

## 3. 栈的应用练习

通过两个练习题，你或许能够明白我前面所说的站在栈的肩膀上思考问题显然要比站在数组的肩膀上思考问题更方便。

### 3.1 合法括号

#### 3.1.1 题目要求

下面的字符串中包含小括号，请编写一个函数判断字符串中的括号是否合法，所谓合法，就是括号成对出现

```
sdf(ds(ew(we)rw)rwqq)qwewe    合法
(sd(qwqw)sd(sd))              合法
()()sd()(sd()fw))(            不合法
```

#### 3.1.2 思路分析

括号存在嵌套关系，也存在并列关系，如果是用数组存储这些括号，然后再想办法一对一对的抵消掉，似乎是一个可行的办法，可是如何判断一个左括号对应的是哪个右括号呢？站在数组的肩膀上思考这个问题，就陷入到一种无从下手的绝望之中。

现在，我们站在栈的肩膀上思考这个问题，解题的步骤就非常简单，我们可以使用for循环遍历字符串的每一个字符，对每个字符做如下的操作：

- 遇到左括号，就把左括号压如栈中
- 遇到右括号，判断栈是否为空，为空说明没有左括号与之对应，缺少左括号，字符串括号不合法，如果栈不为空，则把栈顶元素移除，这对括号抵消掉了

当for循环结束之后，如果栈是空的，就说明所有的左右括号都抵消掉了，如果栈里还有元素，则说明缺少右括号，字符串括号不合法。

#### 3.1.3 示例代码

```
function is_leagl_brackets(string){
    var stack = new Stack();
    for(var i=0; i<string.length; i++ ){
```

```
        var item = string[i];
        if(item == "("){
            // 将左括号压入栈
            stack.push(item);
        }else if (item==")"){
            // 如果为空,就说明没有左括号与之抵消
            if(stack.isEmpty()){
                return false;
            }else{
                // 将栈顶的元素弹出
                stack.pop();
            }
        }

    }
    return stack.size() == 0;
};

console.log(is_leagl_brackets("()()))"));
console.log(is_leagl_brackets("sdf(ds(ew(we)rw)rwqq)qwewe"));
console.log(is_leagl_brackets("()()sd()(sd()fw))("));
```

#### 3.1.4 小结

栈的底层是不是使用了数组这不重要，重要的是栈的这种后进先出的特性，重要的是你只能操作栈顶元素的的限制，一定要忽略掉栈的底层如何实现，而只去关心栈的特性。

我们在编辑文档时，包括写代码，经常进行回退的操作，control+z就可以了，那么你有没有想过，这其实就可以用栈来实现，每一步操作都放入到栈中，当你想回退的时候，就使用pop方法把栈顶元素弹出，于是得到了你之前的一步操作。

### 3.2 计算逆波兰表达式

#### 3.2.1 题目要求

逆波兰表达式，也叫后缀表达式，它将复杂表达式转换为可以依靠简单的操作得到计算结果的表达式，例如 (a+b)*(c+d)转换为ab+cd+*。
示例：

```
["4", "13", "5", "/", "+"] 等价于(4 + (13 / 5)) = 6
["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"] 等价于((10 *
(6 / ((9 + 3) * -11))) + 17) + 5
```

请编写函数calc_exp(exp) 实现逆波兰表达式计算，exp的类型是数组。

### 3.2.1 思路分析

["4", "13", "5", "/", "+"] 就是一个数组，在数组层面上思考这个问题，遇到 / 时，把13 和 5 拿出来计算，然后把13 和 5 删除并把结果放入到4的后面，天哪，太复杂了，太笨拙了，我已经无法继续思考了。

如果是使用栈来解决这个问题，一切就那么的自然而简单，使用for循环遍历数组，对每一个元素做如下操作：

- 如果元素不是 + - * / 中的某一个，就压入栈中
- 如果元素是 + - * / 中的某一个，则从栈里连续弹出两个元素，并对这两个元素进行计算，将计算结果压入栈中

for循环结束之后，栈里只有一个元素，这个元素就是整个表达式的计算结果

### 3.2.2 示例代码

```
function calc_exp(exp){
    var stack = new Stack();
    for(var i = 0; i < exp.length;i++){
        var item = exp[i];

        if(["+", "-", "*", "/"].indexOf(item) >= 0){
            // 从栈顶弹出两个元素
            var value_1 = stack.pop();
            var value_2 = stack.pop();
            // 拼成表达式
            var exp_str = value_2 + item + value_1;
            // 计算并取整
            var res = parseInt(eval(exp_str));
            // 将计算结果压如栈
            stack.push(res.toString());
        }else{
            stack.push(item);
        }
    }
    // 表达式如果是正确的,最终,栈里还有一个元素,且正是表达式的计算结果
    return stack.pop();
};


var exp_1 = ["4", "13", "5", "/", "+"];
```

```
var exp_2 = ["10", "6", "9", "3", "+", "-11", "*", "/", "*", "17", "+", "5", "+"];
console.log(calc_exp(exp_1));
console.log(calc_exp(exp_2));
```

## 4. 课后练习题

### 4.1 实现一个有min方法的栈

实现一个栈，除了常见的push，pop方法以外，提供一个min方法，返回栈里最小的元素，且时间复杂度为o(1)

### 思路分析

使用两个栈来存储数据，其中一个命名为data_stack,专门用来存储数据，另一个命名为min_stack，专门用来存储栈里最小的数据。

注意了，我接下来的分析过程非常重要，我希望你能多阅读几遍。

1. 我们要实现的是一个栈，除了常规的方法，还要有一个min方法，data_stack就是专门为常规方法而存在的，min_stack就是为了这个min方法而存在的。

2. 编程思想里有一个分而治之的思想，简单来说，就是分开想，分开处理。那么我们现在考虑data_stack，这个时候别管min方法，你就只关心data_stack，它就是一个普通的栈啊，没什么特别的，一个简单的栈你还不会么，就是push，pop那些方法，正常实现就可以了。

   data_stack处理完了以后，再考虑min_stack，这个时候，你就别想data_stack了，只关心min_stack，它要存储栈里的最小值，我们先考虑边界情况，如果min_stack为空，这个时候，如果push进来一个数据，那这个数据一定是最小的，所以此时，直接放入min_stack即可。如果min_stack不为空，这个时候它的栈顶不正是栈的最小元素么，如果push进来的元素比栈顶元素还小，放入min_stack就好了，这样，min_stack的栈顶始终都是栈里的最小值。

示例代码:

```
function MinStack(){
    var data_stack = new Stack();    // 普通的栈
    var min_stack = new Stack();     // 专门存储最小值

    // push的时候,两个栈都要操作
    this.push = function(item){
        data_stack.push(item);     // data_stack是常规栈,常规操作即可
```

```
        // 如果min_stack为空,直接放入,如果item小于min_stack栈顶元素,放入其中
        // 这样做的目的,是保证min_stack的栈顶始终保存栈的最小值
        if(min_stack.isEmpty() || item < min_stack.top()){
            min_stack.push(item);
        }else{
            // 如果item大于等于栈顶元素,把min_stack的栈顶元素再放入一次
            // min_stack的元素个数要和data_stack 保持一致
            min_stack.push(min_stack.top());
        }

    };

    // pop的时候,两个栈都pop
    this.pop = function(){
        min_stack.pop();
        return data_stack.pop();
    };

    // 直接取栈顶的元素
    this.min = function(){
        return min_stack.top();
    };
};


minstack = new MinStack();

minstack.push(3);
minstack.push(6);
minstack.push(8);
console.log(minstack.min());
minstack.push(2);
console.log(minstack.min());
```

### 4.2 使用栈，完成中序表达式转后续表达式

例如

```
输入:["12","+", "3"]
输出:["12","3","+"]

输入:["(","1","+","(","4","+","5","+","3",")",
"-","3",")","+","(","9","+","8",")"]
输出:[ '1', '4', '5', '+', '3', '+', '+', '3', '-', '9', '8', '+', '+' ]
```

```
输入:['(', '1', '+', '(', '4', '+', '5', '+', '3',
')', '/', '4', '-', '3', ')', '+', '(', '6', '+', '8', ')', '*', '3']
输出:['1', '4', '5', '+', '3', '+', '4',
'/', '+', '3', '-', '6', '8', '+', '3', '*', '+']
```

### 思路分析

定义数组postfix_lst，用于存储后缀表达式，遍历中缀表达式，对每一个遍历到的元素做如处理:
1、如果是数字,直接放入到postfix_lst中
2、遇到左括号入栈
3、遇到右括号,把栈顶元素弹出并放入到postfix_lst中,直到遇到左括号，最后弹出左括号
4、遇到运算符,把栈顶的运算符弹出,直到栈顶的运算符优先级小于当前运算符， 把弹出的运算符加入到postfix_lst，当前的运算符入栈
5、for循环结束后, 栈里可能还有元素,都弹出放入到postfix_lst中

```
// 定义运算符的优先级
var priority_map = {
    "+": 1,
    "-": 1,
    "*": 2,
    "/": 2};

function infix_exp_2_postfix_exp(exp){
    var stack = new Stack();

    var postfix_lst = [];
    for(var i = 0;i<exp.length;i++){
        var item = exp[i];
        // 如果是数字,直接放入到postfix_lst中
        if(!isNaN(item)){
            postfix_lst.push(item);
        }else if (item == "("){
            // 遇到左括号入栈
            stack.push(item);
        }else if (item == ")"){
            // 遇到右括号,把栈顶元素弹出,直到遇到左括号
            while(stack.top() != "("){
                postfix_lst.push(stack.pop());
            }
            stack.pop();   // 左括号出栈
        }else{
```

```
            // 遇到运算符,把栈顶的运算符弹出,直到栈顶的运算符优先级小于当前运算符
            while(!stack.isEmpty() && ["+", "-", "*", "/"].indexOf(stack.top()) >=
 0 && priority_map[stack.top()] >= priority_map[item]){
                // 把弹出的运算符加入到postfix_lst
                postfix_lst.push(stack.pop());
            }
            // 当前的运算符入栈
            stack.push(item);
        }

    }

    // for循环结束后, 栈里可能还有元素,都弹出放入到postfix_lst中
    while(!stack.isEmpty()) {
        postfix_lst.push(stack.pop())
    }

    return postfix_lst
};


// 12+3
console.log(infix_exp_2_postfix_exp(["12","+", "3"]))
// 2-3+2
console.log(infix_exp_2_postfix_exp(["2","-", "3", "+", "2"]))
// (1+(4+5+3)-3)+(9+8)
var exp = ["(","1","+","(","4","+","5","+","3",")","-","3",")","+","(","9","+","8"
,")"];
console.log(infix_exp_2_postfix_exp(exp))

// (1+(4+5+3)/4-3)+(6+8)*3
var exp = ['(', '1', '+', '(', '4', '+', '5', '+', '3', ')', '/', '4', '-', '3', '
)', '+', '(', '6', '+', '8', ')', '*', '3']
console.log(infix_exp_2_postfix_exp(exp))

console.log(infix_exp_2_postfix_exp(["12","+", "3","*", "5"]))
console.log(infix_exp_2_postfix_exp(["12","*", "3","+", "5"]))
```

### 算法思路推理过程

推理的过程要从简入繁，先考虑最简单的情况

#### 1、只有一个运算符

```
中缀：1 + 2
后缀：1 2 +
```

后缀表达式，数值在前，操作符在后，因此，遇到数值时直接放入到后缀表达式中。将操作符放入栈中，等到中缀表达式遍历结束后，将栈里的操作符弹出放入到后缀表达式中

**2、多个运算符，栈顶操作符优先级和当前运算符相同**

多个运算符，似乎也可以像第一步推理中那样操作，但实际不行

```
中缀：1 + 2 - 3
后缀：1 2 + 3 -
```

每次遇到操作符时，如果栈里有操作符，说明前面有需要计算的数值，且计算的操作符就在栈顶，应该弹出放入到后缀表达式中，如果都等到中缀表达式结束再弹出，就会变成 1 2 3 + –

就本示例而言，遇到减号时，后缀表达式里是 1 2 ，栈里是+，1和2需要计算，进行计算的操作符就在栈顶，因此需要弹出，放入到后缀表达式，之后减号入栈，中缀表达式遍历结束后，后缀表达式是 1 2 + 3， 栈里是–，将栈里的操作符去全部弹出，放入到后缀表达式，最终结果为 1 2 + 3 –.

**3、多个运算符，栈顶操作符优先级大于当前运算符相同**

```
中缀：1 * 2 + 3
后缀：1 2 * 3 +
```

栈顶运算符优先级高，和第2步的分析一致，前面有需要计算的数值，应该弹出放入到后缀表达式中

**4、多个运算符，栈顶操作符优先级小于当前运算符相同**

```
中缀：1 + 2 * 3
后缀：1 2 3 * +
```

栈顶运算符优先级低，说明还不能进行计算，要继续压栈，压栈后，高优先级操作符在栈顶，出栈的时候先出，保证2*3先被计算

### 5、有括号的情况

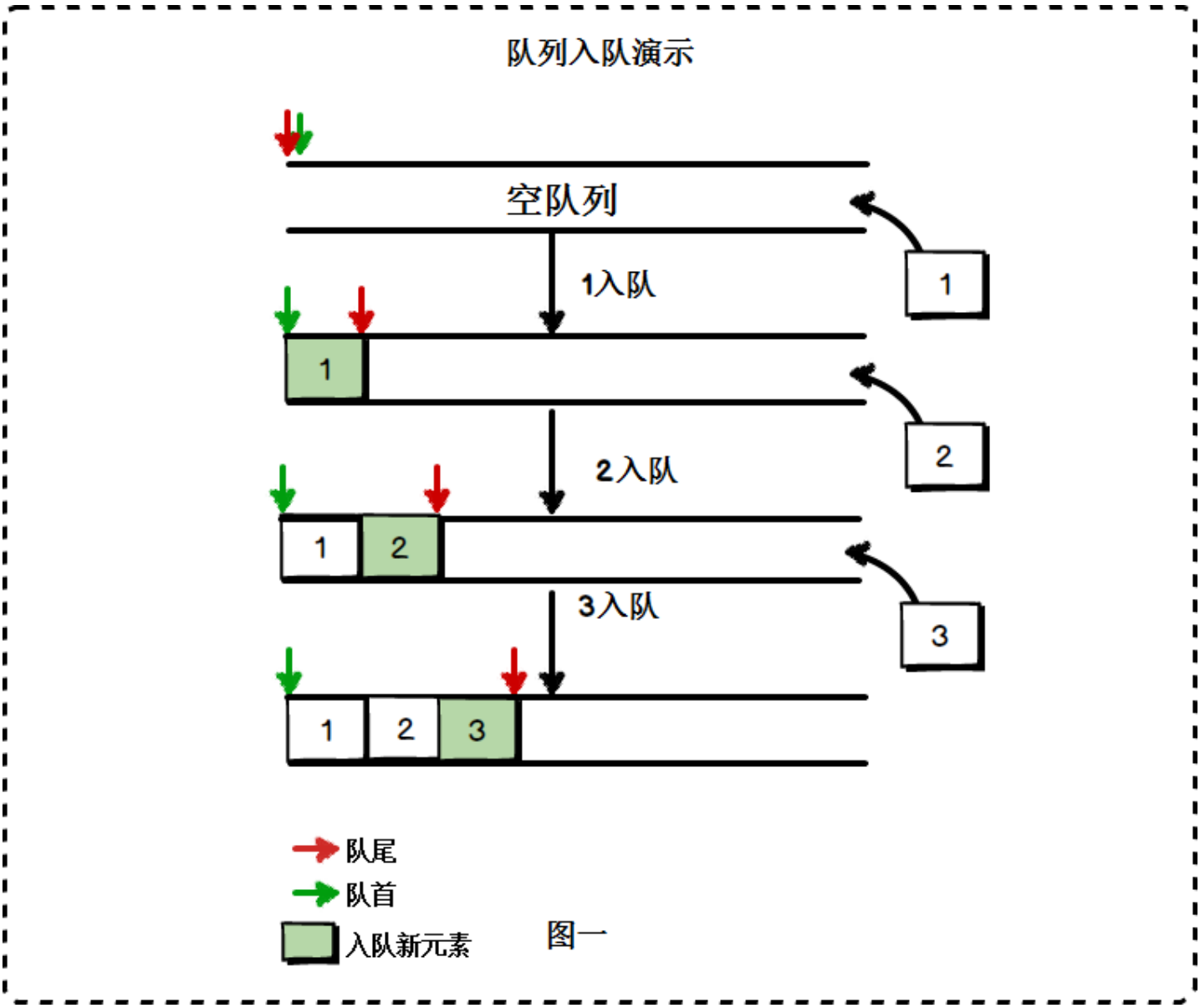
括号里的的表达式可以视为一个独立的中缀表达式，因此，前面4步的分析都适用，但是括号里的中缀表达式在一个更大的表达式中，因此需要与其他部分分隔，分隔的方法就是遇到小括号后压栈，此后的操作遵循前面的推理逻辑，当遇到右括号时，说明括号内的表达式结束了，根据第一步的分析，应该把所有属于这个括号内表达式的操作符都弹出来放入到后缀表达式，最后一步弹出左括号。

## 数据结构之----队列

### 1、队列的定义

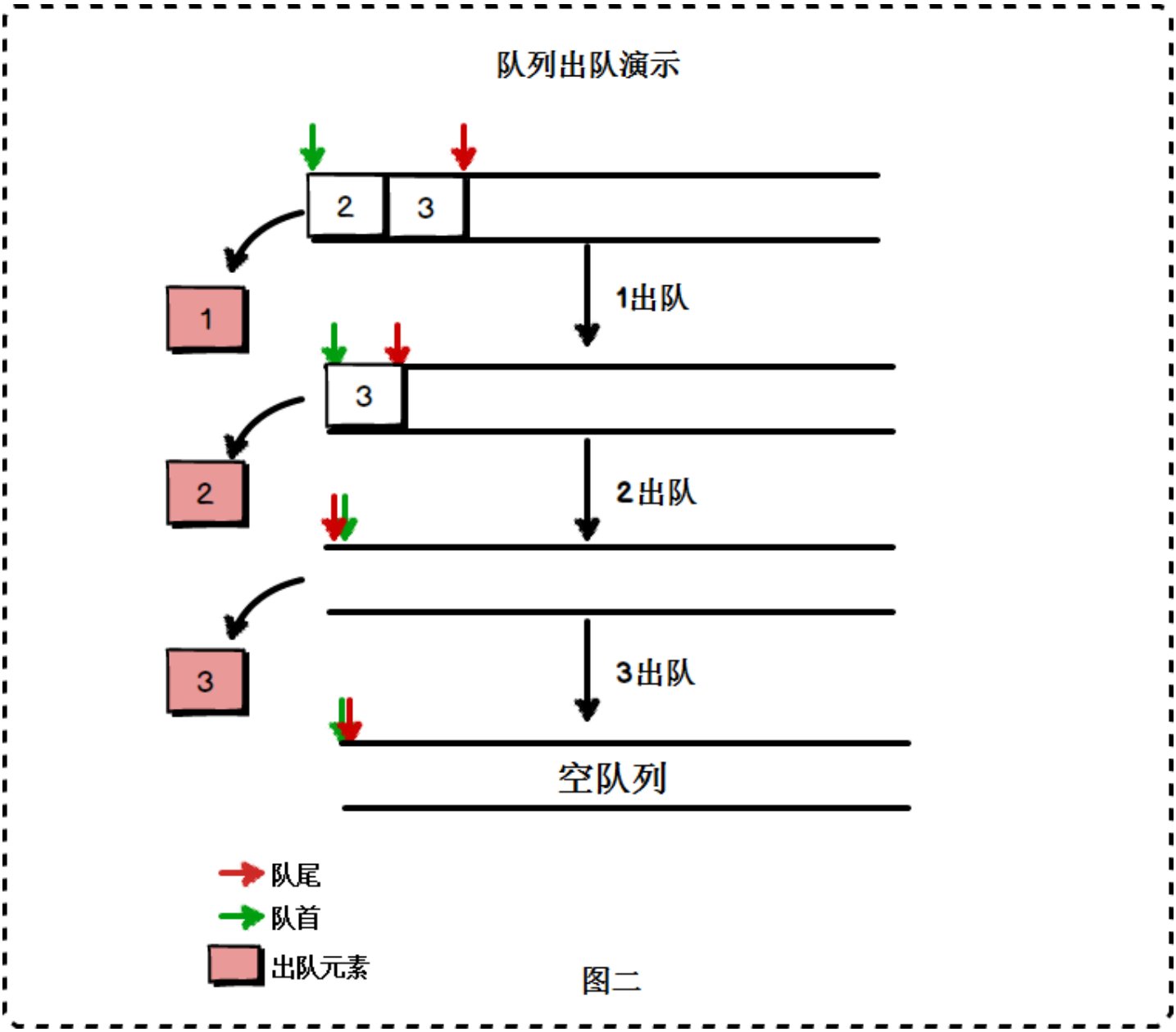
队列是一种特殊的线性表，其特殊之处在于，它只允许你在队列的头部删除元素，在队列的末尾添加新的元素。

下面这张图展示了队列如何添加新的元素




图一

左侧是队列的头部，右侧是队列的尾部，新的元素如果想进入队列，只能从尾部进入，如果想要出队列，只能从队列的头部出去,下面的图展示了元素如何出队列




图二

日常生活中，排队就是典型的队列结构，下面这幅图里，大家在排队办理登机手续。

## 2. 队列的实现

有了栈这个数据结构做铺垫，队列就容易学了。

### 2.1 数据存储

同栈一样，本课程里队列的实现也使用数组来存储数据，定义一个简单的Queue类

```
function Queue(){
    var items = [];   // 存储数据
};
```

数据将存储在items数组之中，现在，这个类没有任何方法。

### 2.2 队列的方法

队列的方法如下:

- enqueue 从队列尾部添加一个元素（新来一个排队的人，文明礼貌，站在了队伍末尾）
- dequeue 从队列头部删除一个元素（排队伍最前面的人刚办理完登机手续，离开了队伍）
- head 返回头部的元素，注意，不是删除（只是看一下，谁排在最前面）
- size 返回队列大小（数一数有多少人在排队）

- clear 清空队列（航班取消，大家都散了吧）
- isEmpty 判断队列是否为空 （看看是不是有人在排队）
- tail 返回队列尾节点

下面，逐一实现这些方法：

#### 2.2.1 enqueue方法

```
    // 向队列尾部添加一个元素
    this.enqueue = function(item){
        items.push(item);
    };
```

注意只能从尾部添加新元素

#### 2.2.2 dequeue方法

```
    // 移除队列头部的元素
    this.dequeue = function(){
        return items.shift();
    };
```

数组的shift方法从最左侧删除一个元素，实现了队列只能在头部删除数据的要求

#### 2.2.3 head方法

```
    // 返回队列头部的元素
    this.head = function(){
        return items[0];
    }
```

很多实现喜欢用front来命名方法，我自己更喜欢用head

#### 2.2.4 size方法

```
    // 返回队列大小
    this.size = function(){
        return items.length;
    }
```

### 2.2.5 clear方法

```
    // clear
    this.clear = function(){
        items = [];
    }
```

### 2.2.6 isEmpty方法

```
    // isEmpty 判断是否为空队列
    this.isEmpty = function(){
        return items.length == 0;
    }
```

### 2.2.7 tail方法

```
    // 返回队列尾部的元素
    this.tail = function(){
        return items[items.length-1];
    };
```

完整的代码如下：

```
function Queue(){
    var items = [];   // 存储数据

    // 向队列尾部添加一个元素
    this.enqueue = function(item){
        items.push(item);
    };

    // 移除队列头部的元素
    this.dequeue = function(){
        return items.shift();
    };

    // 返回队列头部的元素
    this.head = function(){
```

```
        return items[0];
    }

    // 返回队列大小
    this.size = function(){
        return items.length;
    }

    // clear
    this.clear = function(){
        items = [];
    }

    // isEmpty 判断是否为空队列
    this.isEmpty = function(){
        return items.length == 0;
    }
};
```

## 3. 队列的应用练习

栈的特性是是先进后出，队列的特性是先进先出。

### 3.1 约瑟夫环（普通模式）

#### 3.1.1 题目要求

有一个数组a[100]存放0--99;要求每隔两个数删掉一个数，到末尾时循环至开头继续进行，求最后一个被删掉的数。

#### 3.1.2 思路分析

前10个数是 0 1 2 3 4 5 6 7 8 9 10，所谓每隔两个数删掉一个数，其实就是把 2 5 8 删除掉，如果只是从0 到 99 每个两个数删掉一个数，其实挺简单的，可是题目要求到末尾时还有循环至开头继续进行。

如果是用数组，问题就又显得麻烦了，关键是到了末尾如何回到开头重新来一遍，还得考虑把删除掉的元素从数组中删除。

如果用队列就简单了，先将这100个数放入队列，使用while循环，while循环终止的条件是队列里只有一个元素。使用index变量从0开始计数，算法步骤如下:

1. 从队列头部删除一个元素，index+1

2. 如果index%3 == 0,就说明这个元素是需要删除的元素，如果不等于0，就不是需要被删除的元素，则把它添加到队列的尾部

不停的有元素被删除，最终队列里只有一个元素，此时while循环终止，队列的所剩的元素就是最后一个被删除的元素。

#### 3.1.3 示例代码

```
function del_ring(arr_list){
    // 把数组里的元素都放入到队列中
    var queue = new Queue();
    for(var i=0;i< arr_list.length;i++){
        queue.enqueue(arr_list[i]);
    }

    var index = 0;
    while(queue.size() != 1){
        // 弹出一个元素,判断是否需要删除
        var item = queue.dequeue();
        index += 1;
        // 每隔两个就要删除掉一个,那么不是被删除的元素就放回到队列尾部
        if(index %3 != 0){
            queue.enqueue(item);
        }
    }

    return queue.head();
};

// 准备好数据
var arr_list = [];
for(var i=0;i< 100;i++){
    arr_list.push(i);
}


console.log(del_ring(arr_list));
```

### 3.2 斐波那契数列（普通模式）

斐波那契数列是一个非常经典的问题，有着各种各样的解法，比较常见的是递归算法，其实也可以使用队列来实现

### 3.2.1 题目要求

使用队列计算斐波那契数列的第n项

### 3.2.2 思路分析

斐波那契数列的前两项是 1 1 ，此后的每一项都是该项前面两项之和，即f(n) = f(n−1) + f(n−2)。

如果使用数组来实现，究竟有多麻烦了我就不赘述了，直接考虑使用队列来实现。

先将两个1 添加到队列中，之后使用while循环，用index计数，循环终止的条件是index < n −2

- 使用dequeue方法从队列头部删除一个元素，该元素为del_item
- 使用head方法获得队列头部的元素，该元素为 head_item
- del_item + head_item = next_item,将next_item放入队列，注意，只能从尾部添加元素
- index+1

当循环结束时，队列里面有两个元素，先用dequeue 删除头部元素，剩下的那个元素就是我们想要的答案

### 3.2.3 示例代码

```
function fibonacci(n){
    queue = new Queue();
    var index = 0;
    // 先放入斐波那契序列的前两个数值
    queue.enqueue(1);
    queue.enqueue(1);
    while(index < n-2){
        // 出队列一个元素
        var del_item = queue.dequeue();
        // 取队列头部元素
        var head_item = queue.head();
        var next_item = del_item + head_item;
        // 将计算结果放入队列
        queue.enqueue(next_item);
        index += 1;
    }

    queue.dequeue();
    return queue.head();
```

```
};


console.log(fibonacci(8));
```

### 3.2.4 小结

使用队列的例子还有很多，比如逐层打印一颗树上的节点。像kafka，rabbitmq这类消息队列，其形式就是一种队列，消息生产者把消息放入队列中（尾部），消费者从队列里取出消息进行处理（头部），只不过背后的实现更为复杂。

如果你了解一点socket，那么你应该知道当大量客户端向服务端发起连接，而服务端忙不过来的时候，就会把这些请求放入到队列中，先来的先处理，后来的后处理，队列满时，新来的请求直接抛弃掉。

数据结构在系统设计中的应用非常广泛，只是我们水平达不到那个级别，知道的太少，但如果能理解并掌握这些数据结构，那么就有机会在工作中使用它们并解决一些具体的问题，当我们手里除了锤子还有电锯时，那么我们的眼里就不只是钉子，解决问题的思路也会更加开阔。

## 3.3 用队列实现栈（困难模式）

### 3.3.1 题目要求

用两个队列实现一个栈

### 3.3.2 思路分析

队列是先进先出，而栈是先进后出，两者对数据的管理模式刚好是相反的，但是却可以用两个队列实现一个栈。

两个队列分别命名为queue_1, queue_2,实现的思路如下:

- push， 实现push方法时，如果两个队列都为空，那么默认向queue_1里添加数据，如果有一个不为空，则向这个不为空的队列里添加数据
- top，两个队列，或者都为空，或者有一个不为空，只需要返回不为空的队列的尾部元素即可
- pop，pop方法是比较复杂，pop方法要删除的是栈顶，但这个栈顶元素其实是队列的尾部元素。每一次做pop操作时，将不为空的队列里的元素一次删除并放入到另一个队列中直到遇到队列中只剩下一个元素，删除这个元素，其余的元素都跑到之前为空的队列

中了。

在具体的实现中，我定义额外的两个变量，data_queue和empty_queue，data_queue始终指向那个不为空的队列，empty_queue始终指向那个为空的队列。

### 3.3.3 代码示例

```
function QueueStack(){
    var queue_1 = new Queue();
    var queue_2 = new Queue();
    var data_queue = null;      // 放数据的队列
    var empty_queue = null;     // 空队列,备份使用

    // 确认哪个队列放数据,哪个队列做备份空队列
    var init_queue = function(){
        // 都为空,默认返回queue_1
        if(queue_1.isEmpty() && queue_2.isEmpty()){
            data_queue = queue_1;
            empty_queue = queue_2;
        }else if(queue_1.isEmpty()){
            data_queue = queue_2;
            empty_queue = queue_1;
        }else{
            data_queue = queue_1;
            empty_queue = queue_2;
        }
    };


    // push方法
    this.push = function (item) {
        init_queue();
        data_queue.enqueue(item);
    };

    // top方法
    this.top = function(){
        init_queue();
        return data_queue.tail();
    }

    /**
     * pop方法要弹出栈顶元素,这个栈顶元素,其实就是queue的队尾元素
     * 但是队尾元素是不能删除的,我们可以把data_queue里的元素(除了队尾元素)都移除放入到em
pty_queue中
```

```
     * 最后移除data_queue的队尾元素并返回
     * data_queue 和 empty_queue 交换了身份
     */
    this.pop = function(){
        init_queue();
        while(data_queue.size()>1){
            empty_queue.enqueue(data_queue.dequeue());
        }
        return data_queue.dequeue();
    };
};


var q_stack = new QueueStack();
q_stack.push(1);
q_stack.push(2);
q_stack.push(4);
console.log(q_stack.top());   // 栈顶是 4
console.log(q_stack.pop());   // 移除 4
console.log(q_stack.top());   // 栈顶变成 2
console.log(q_stack.pop());   // 移除 2
console.log(q_stack.pop());   // 移除 2
```

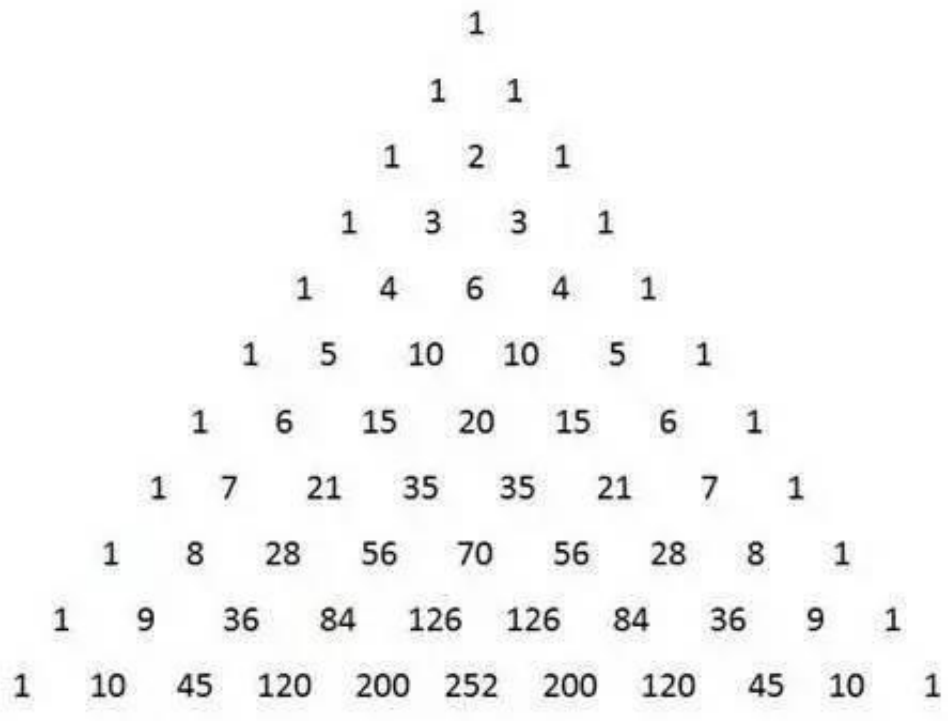
### 3.4 打印杨辉三角（困难模式）

#### 3.4.1 题目要求

使用队列打印出杨辉三角的前n行，n >= 1

#### 3.4.2 思路分析

杨辉三角示意图如下




杨辉三角中的每一行，都依赖于上一行，假设在队列里存储第n – 1行的数据，输出第n行时，只需要将队列里的数据依次出队列，进行计算得到下一行的数值并将计算所得放入到队列中。

计算的方式：f[i][j] = f[i–1][j–1] + f[i–1][j], i 代表行数，j代表一行的第几个数，如果j= 0 或者 j = i ,则 f[i][j] = 1。

但是将计算所得放入到队列中时，队列中保存的是两行数据，一部分是第n–1行，另一部分是刚刚计算出来的第n行数据，需要有办法将这两行数据分割开。

分开的方式有两种，一种是使用for循环进行控制，在输出第5行时，其实只有5个数据可以输出，那么就可以使用for循环控制调用enqueue的次数，循环5次后，队列里存储的就是计算好的第6行的数据。

第二种方法是每一行的数据后面多存储一个0，使用这个0来作为分界点，如果enqueue返回的是0，就说明这一行已经全部输出，此时，将这个0追加到队列的末尾。

### 3.4.3 示例代码

```
function print_yanghui(n){
    var queue = new Queue();
    queue.enqueue(1);
    // 第一层for循环控制打印几层
    for(var i=1; i<=n; i++){
        var line = "";
        var pre = 0;
```

```
        // 第二层for循环控制打印第 i 层
        for(var j=0; j<i; j++){
            var item = queue.dequeue();
            line += item + "  "
            // 计算下一行的内容
            var value = item + pre;
            pre = item;
            queue.enqueue(value);
        }
        // 每一层最后一个数字是1,上面的for循环没有计算最后一个数
        queue.enqueue(1);
        console.log(line);
    }
};

function print_yanghui_2(n){
    var queue = new Queue();
    queue.enqueue(1);
    queue.enqueue(0);
    for(var i=1; i<=n; i++){
        var line = "";
        var pre = 0;
        while(true){
            var item = queue.dequeue();
            // 用一个0把每一行的数据分割开,遇到0不输出,
            if(item==0){
                queue.enqueue(1);
                queue.enqueue(0);
                break
            }else {
                // 计算下一行的内容
                line += item + "  "
                var value = item + pre;
                pre = item;
                queue.enqueue(value);
            }
        }
        console.log(line);
    }
}


print_yanghui(10);
print_yanghui_2(10);
```

## 4 课后作业

### 4.1.1 用两个栈实现一个队列（普通模式）

栈是先进后出，队列是先进先出，但可以用两个栈来模拟一个队列，请实现enqueue，dequeue， head这三个方法。

### 4.1.2 思路分析

我们所学的每一种数据结构，本质上研究的是对数据如何存储和使用，这就必然涉及到增加，删除，修改，获取这四个操作，日常工作其实所作的也逃不掉这四种业务。

那么在考虑实现这些方法时，我们优先考虑如何实现增加，道理很简单，只有先增加，才能继续研究如何删除，修改，获取，否则，没有数据，怎么研究？

就这道题目而言，我们先考虑如何实现enqueue方法，两个栈分别命名为stack_1,stack_2,面对这两个栈，你能怎么做呢，似乎也只好选一个栈用来存储数据了，那就选stack_1来存储数据吧，看起来是一个无奈之举，但的确实现了enqueue方法。

接下来考虑dequeue方法，队列的头，在stack_1的底部，栈是先进后出，目前取不到，可不还有stack_2么，把stack_1里的元素都倒入到stack_2中，这样，队列的头就变成了stack_2的栈顶，这样不就可以执行stack_2.pop()来删除了么。执行完pop后，需要把stack_2里的元素再倒回到stack_1么，不需要，现在队列的头正好是stack_2的栈顶，恰好可以操作，队列的dequeue方法借助栈的pop方法完成，队列的head方法借助栈的top方法完成。

如果stack_2是空的怎么办？把stack_1里的元素都倒入到stack_2就可以了，这样，如果stack_1也是空的，说明队列就是空的，返回null就可以了。

enqueue始终都操作stack_1，dequeue和head方法始终都操作stack_2。

### 4.1.3 思路分析

```
function StackQueue(){
    var stack_1 = new Stack();
    var stack_2 = new Stack();

    // 总是把数据放入到stack_1中
    this.enqueue = function(item){
        stack_1.push(item);
    };

    // 获得队列的头
    this.head = function(){
```

```
        // 两个栈都是空的
        if(stack_2.isEmpty() && stack_1.isEmpty()){
            return null;
        }

        // 如果stack_2 是空的,那么stack_1一定不为空,把stack_1中的元素倒入stack_2
        if(stack_2.isEmpty()){
            while(!stack_1.isEmpty()){
                stack_2.push(stack_1.pop());
            }
        }
        return stack_2.top();
    };

    // 出队列
    this.dequeue = function(){
        // 两个栈都是空的
        if(stack_2.isEmpty() && stack_1.isEmpty()){
            return null;
        }

        // 如果stack_2 是空的,那么stack_1一定不为空,把stack_1中的元素倒入stack_2
        if(stack_2.isEmpty()){
            while(!stack_1.isEmpty()){
                stack_2.push(stack_1.pop());
            }
        }
        return stack_2.pop();
    };

};


var sq = new StackQueue();
sq.enqueue(1);
sq.enqueue(4);
sq.enqueue(8);
console.log(sq.head());
sq.dequeue();
sq.enqueue(9);
console.log(sq.head());
sq.dequeue();
console.log(sq.head());
console.log(sq.dequeue());
console.log(sq.dequeue());
```

## 4.2 迷宫问题（地狱模式）

### 4.2.1 题目要求

有一个二维数组

```
var maze_array = [[0, 0, 1, 0, 0, 0, 0],
                  [0, 0, 1, 1, 0, 0, 0],
                  [0, 0, 0, 0, 1, 0, 0],
                  [0, 0, 0, 1, 1, 0, 0],
                  [1, 0, 0, 0, 1, 0, 0],
                  [1, 1, 1, 0, 0, 0, 0],
                  [1, 1, 1, 0, 0, 0, 0]
                 ];
```

元素为0，表示这个点可以通行，元素为1，表示不可以通行，设置起始点为maze_array[2][1]，终点是maze_array[3][5]，请用程序计算这两个点是否相通，如果相通请输出两点之间的最短路径（从起始点到终点所经过的每一个点）

### 4.3.2 思路分析

从maze_array[2][1]开始，把这个点能到达的邻近点都标记为1（表示与起始点距离为1），然后把标记为1的点能够到达的邻近点标记为2（表示与起始点距离为2），如此继续处理，直到到达终点，或者找不到可以到达的邻近点为止。标记后的结构图如下

```
[ 3, 2, 0, 0, 0, 0, 0 ]
[ 2, 1, 0, 0, 0, 0, 0 ]
[ 1, 0, 1, 2, 0, 0, 0 ]
[ 2, 1, 2, 0, 0, 0, 0 ]
[ 0, 2, 3, 4, 0, 8, 0 ]
[ 0, 0, 0, 5, 6, 7, 8 ]
[ 0, 0, 0, 6, 7, 8, 0 ]
```

从起始点到终点，需要经过8个点，这是最短的连通路径。这时，要从终点开始反方向寻找路径，在终点的四周，一定存在一个点被标记为8，这个标记为8的点的四周一定存在一个点被标记为7，以此类推，最终找到标记为1的那个点，这个点的四周一定有一个点是起始点。

### 4.3.3 示例代码

```
// 起始点是maze_array[2][1], 终点是 maze_array[3][5]
var maze_array = [[0, 0, 1, 0, 0, 0, 0],
```

```
    [0, 0, 1, 1, 0, 0, 0],
    [0, 0, 0, 0, 1, 0, 0],
    [0, 0, 0, 1, 1, 0, 0],
    [1, 0, 0, 0, 1, 0, 0],
    [1, 1, 1, 0, 0, 0, 0],
    [1, 1, 1, 0, 0, 0, 0]
];


var Node = function(x, y){
    this.x = x;
    this.y = y;
    this.step = 0;
};

var Position = function(x, y){
    this.x = x;
    this.y = y;
}


// 找到pos可以到达的点
function find_position(pos, maze){
    var x = pos.x;
    var y = pos.y;
    var pos_arr = [];
    // 上面的点
    if(x-1 >= 0){
        pos_arr.push(new Position(x-1, y));
    }
    // 右面的点
    if(y+1 < maze[0].length){
        pos_arr.push(new Position(x, y+1));
    }
    // 下面的点
    if(x+1 < maze.length){
        pos_arr.push(new Position(x+1, y));
    }
    // 左面的点
    if(y-1 >= 0){
        pos_arr.push(new Position(x, y-1));
    }
    return pos_arr;
};

function print_node(maze_node){
```

```
    for(var i = 0; i<maze_node.length;i++){
        var arr = [];
        for(var j =0;j<maze_node[i].length;j++){
            arr.push(maze_node[i][j].step);
        }
        console.log(arr);
    }
}

function find_path(maze, start_pos, end_pos){
    var maze_node = [];
    // 初始化maze_node,用于记录距离出发点的距离
    for(var i = 0; i< maze_array.length; i++){
        var arr = maze_array[i];
        var node_arr = [];
        for(var j =0; j< arr.length; j++){
            var node = new Node(i, j);
            node_arr.push(node);
        }
        maze_node.push(node_arr);
    }

    // 先把出发点放入到队列中
    var queue = new Queue();
    queue.enqueue(start_pos);
    var b_arrive = false;
    var max_step = 0;          // 记录从出发点到终点的距离
    while(true){
        // 从队列中弹出一个点,计算这个点可以到达的位置
        var position = queue.dequeue();
        var pos_arr = find_position(position, maze)
        for(var i =0; i<pos_arr.length; i++){
            var pos = pos_arr[i];
            // 判断是否到达终点
            if(pos.x == end_pos.x && pos.y==end_pos.y){
                b_arrive = true;
                max_step = maze_node[position.x][position.y].step;
                break;
            }

            // 起始点
            if(pos.x == start_pos.x && pos.y == start_pos.y){
                continue;
            }
            // 不能通过
            if(maze[pos.x][pos.y]==1){
```

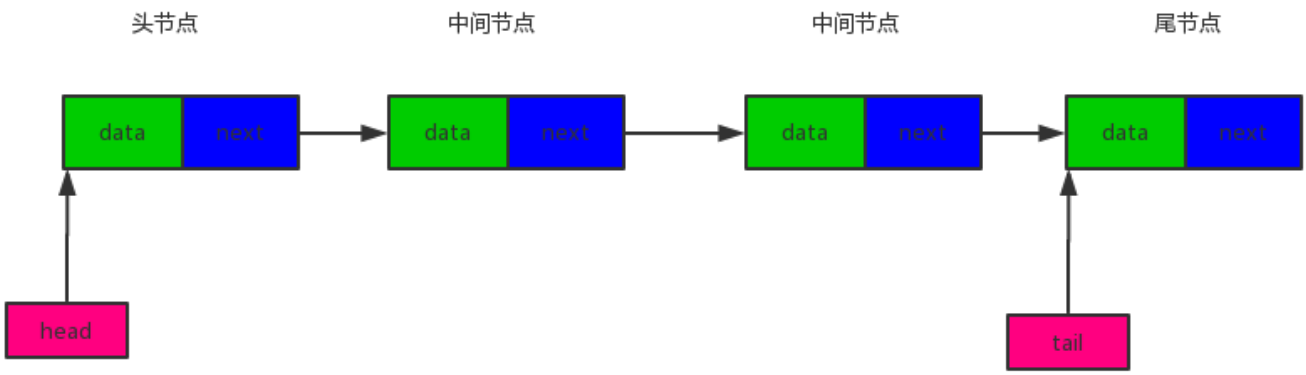
```
                continue;
            }
            // 已经标识过步数
            if(maze_node[pos.x][pos.y].step > 0){
                continue;
            }
            // 这个点的步数加 1
            maze_node[pos.x][pos.y].step = maze_node[position.x][position.y].step
+ 1;
            queue.enqueue(pos);
        }
        //到达终点了
        if(b_arrive){
            break
        }

        // 栈为空,说明找不到
        if(queue.isEmpty()){
            break;
        }
    }

    // 方向查找路径
    var path = [];
    if(b_arrive){
        // 能够找到路径
        path.push(end_pos);
        var old_pos = end_pos;
        var step = max_step;
        while(step >0){
            var pos_arr = find_position(old_pos, maze);
            for(var i =0; i<pos_arr.length; i++) {
                var pos = pos_arr[i];

                if(maze_node[pos.x][pos.y].step == step){
                    step -= 1;
                    old_pos = pos;
                    path.push(pos);
                    break;
                }
            }
        }
        path.push(start_pos);
    }

    console.log(path.reverse());
```

```
};



var start_pos = new Position(2, 1);
var end_pos = new Position(3, 5);

find_path(maze_array, start_pos, end_pos);
```

# 数据结构之----链表

## 1、链表的定义

链表是物理存储单元上非连续的、非顺序的存储结构，由一系列节点组成，下图是一个简单的结构示意图，本节课程所提到的链表，均指单链表。




关于链表，有几个重要的概念要理清。

### 1.1 节点

节点包含两部分，一部分是存储数据元素的数据域，一部分是存储指向下一个节点的指针域，上图中绿色的部分就是数据域，蓝色的部分是指针域，他们一起共同构成一个节点。一个节点可以用如下的方式去定义和使用，示例代码如下:

```
var Node = function(data){
    this.data = data;
    this.next = null;
}
```

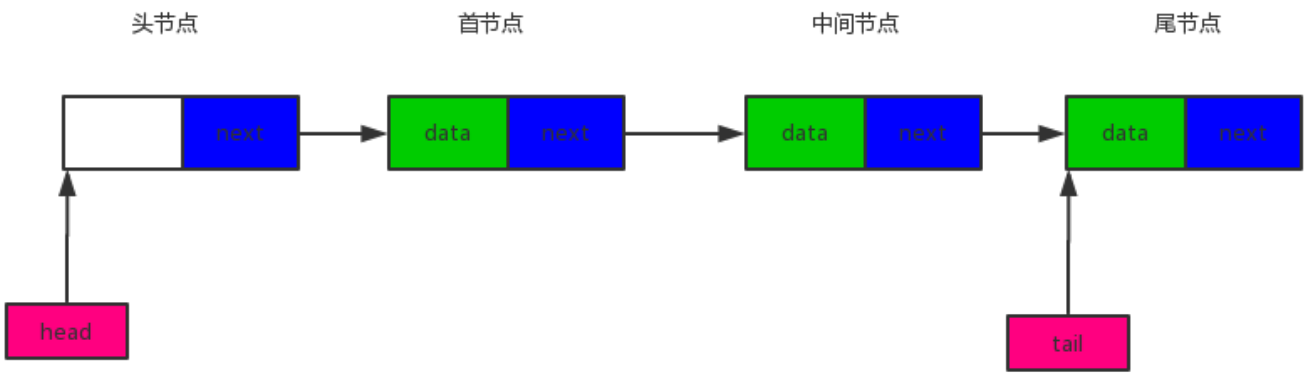
```
var node1 = new Node(1);
var node2 = new Node(2);
var node3 = new Node(3);


node1.next = node2;
node2.next = node3

console.log(node1.data);
console.log(node1.next.data);
console.log(node1.next.next.data);
```

### 1.2 首尾节点

链表中的第一个节点是首节点，最后一个节点是尾节点。

### 1.3 有头链表和无头链表

无头链表是指第一个节点既有数据域，又有指针域，第一个节点既是首节点又是头节点。
有头链表是指第一个节点只有指针域，而没有数据域。
在链表定义中展示的就是无头链表，一个有头链表的结构图如下




本教程采用的是无头链表。

### 1.4 猴子捞月

猴子捞月的故事，大家都一定听过，一只猴子抓住另一只猴子，这就是一个典型的链表，每只猴子都是一个节点。

## 2、链表的实现

### 2.1 定义链表类

先看代码再解释

```
// 定义链表类
function LinkList() {
    // 定义节点
    var Node = function (data) {
        this.data = data;
        this.next = null;
    }

    var length = 0;         // 长度
    var head = null;        // 头节点
    var tail = null;        // 尾节点
};
```

这里，我定义了head和tail，当链表为空时，他们都是null,新增节点后，head指向首节点，tail指向尾节点。

### 2.2 链表的方法

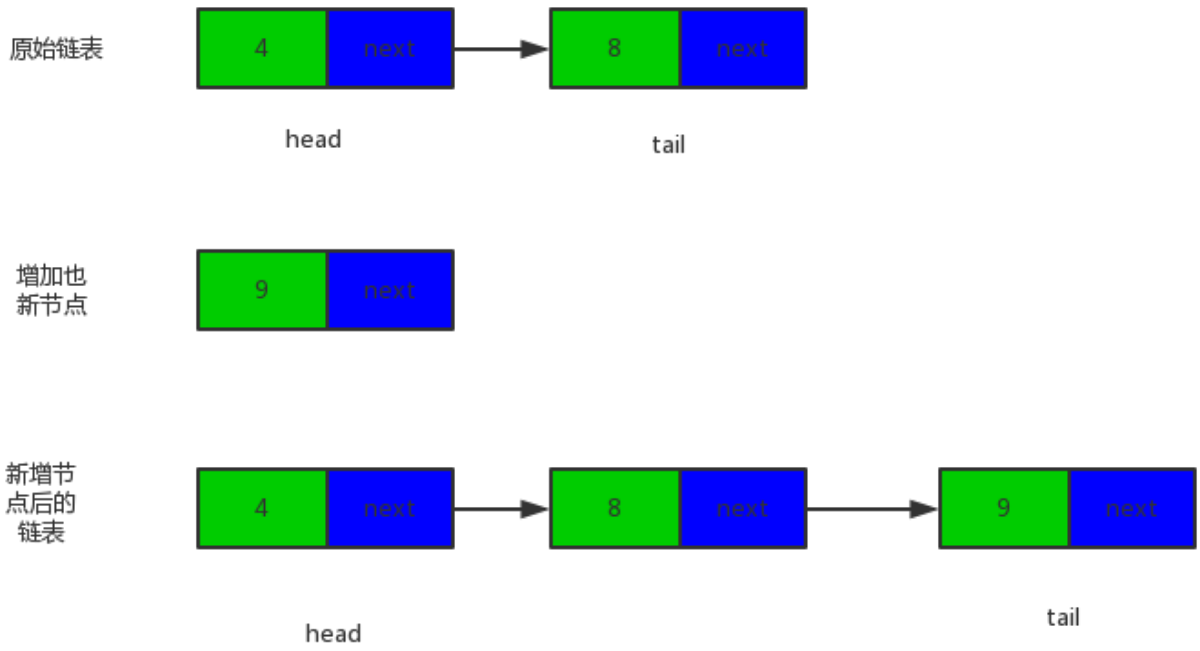
链表的方法如下：

- append， 添加一个新的元素
- insert，在指定位置插入一个元素
- remove，删除指定位置的节点
- remove_head，删除首节点
- remove_tail，删除尾节点
- indexOf，返回指定元素的索引
- get，返回指定索引位置的元素
- head，返回首节点
- tail，返回尾节点
- length，返回链表长度
- isEmpty，判断链表是否为空
- clear，清空链表
- print，打印整个链表

### 2.2.1 append方法

不同于数组实现的stack和queue，这一次，你需要自己来管理每个数据。

- 每次append，都要先创建一个node节点，如果列表为空，则让head和tail指向这个新创建的节点
- 如果列表不为空，则tail.next = node， 并让tail指向node




```
// 添加一个新元素
this.append = function(data){
```

```
        // 创建新节点
        var node = new Node(data);
        // 如果是空链表
        if(head==null){
            head = node;
            tail = head;
        }else{
            tail.next = node;        // 尾节点指向新创建的节点
            tail = node;             // tail指向链表的最后一个节点
        }
        length += 1;                 // 长度加1
        return true;
    };
```

### 2.2.2 print方法

print方法纯粹是为了验证链表其他方法是否正确而存在的，目的是随时可以输出链表，方法虽然简单，但涉及到了链表最重要的操作---遍历链表，我们平时遍历数组时，直接使用for循环，通过索引的变化来获得数组的不同位置的值，但对于链表，我们只能通过next指针找到一个节点的下一个节点。

```
    // 输出链表
    this.print = function(){
        var curr_node = head;
        var str_link = ""
        while(curr_node){
            str_link += curr_node.data.toString() + " ->";
            curr_node = curr_node.next;
        }
        str_link += "null";
        console.log(str_link);
        console.log("长度为"+ length.toString());
    };
```

### 2.2.3 insert方法

append只能在链表的末尾添加元素，而insert可以在指定位置插入一个元素，新增数据的方式更加灵活，insert方法需要传入参数index，指明要插入的索引位置。该方法的关键是找到索引为index-1的节点，只有找到这个节点，才能将新的节点插入到链表中。

如何才能查找到索引为index-1的点呢，我们需要一个方法，这个方法根据索引查找节点并返回，方法定义如下:

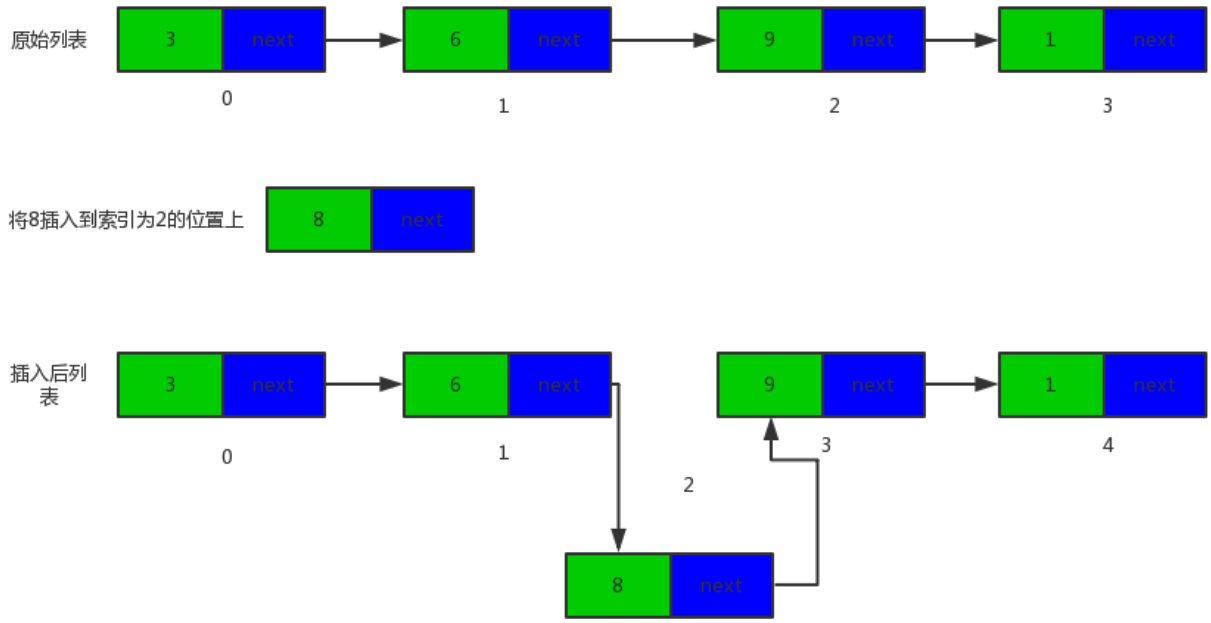
```
var get_node = function(index){
};
```

参考 2.2.2 中的print方法，遍历链表，但是用index来控制遍历的程度，我们让while循环只循环index次就好了，最终方法实现如下:

```
// 获得指定位置的节点
    var get_node = function(index){
        if(index < 0 || index >= length){
            return null;
        }
        var curr_node = head;
        var node_index = index;
        while(node_index-- > 0){
            curr_node = curr_node.next;
        }
        return curr_node;
    };
```

算法思路如下：

- 如果index范围不合法，返回false
- 考虑边界情况，如果index==length，就是在尾部插入，直接调用append方法，如果index==0, 那么这个节点就变成了新的头节点
- 不是边界，找到索引为index-1的节点，用这个节点连接新节点

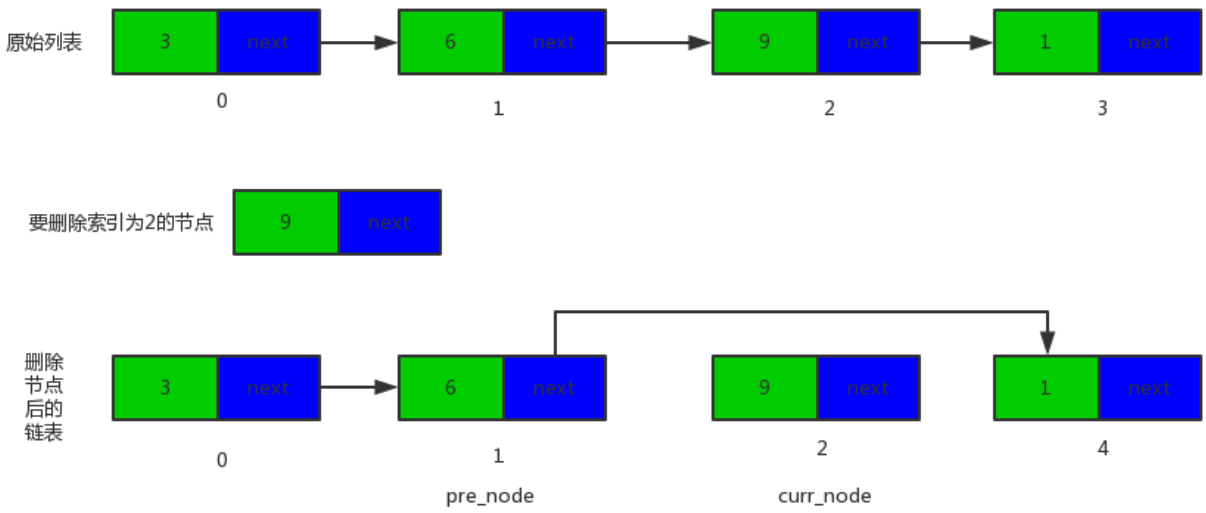



```
// 在指定位置插入新的元素
this.insert = function(index, data){
    // index == length,说明是在尾节点的后面新增,直接调用append方法即可
    if(index == length){
        return this.append(data);
    }else if(index > length || index < 0){
        // index范围错误
        return false;
    }else{
        var new_node = new Node(data);
        if(index == 0){
            // 如果在头节点前面插入,新的节点就变成了头节点
            new_node.next= head;
            head = new_node;
        }else{
            // 要插入的位置是index,找到索引为index-1的节点,然后进行连接
            var pre_node = get_node(index-1);
            new_node.next = pre_node.next;
            pre_node.next = new_node;
        }
        length += 1;
        return true;
    }
};
```

### 2.2.4 remove方法

删除指定位置的节点，需要传入参数index,和insert方法一样，先考虑索引的范围是否合法，然后考虑索引在边界时的操作，关键点是找到索引为index-1的这个节点，这个节点的next指向了要删除的节点。

- 如果index< 或者 index>=length，索引都是错误的，返回null
- 如果index==0,删除的是头节点，只需要执行head = head.next就可以把头节点删除
- 如果index > 0,那么就找到索引为index-1的节点，利用这个节点将索引为index的节点删除
- 删除节点时，如果被删除的节点是尾节点，tail要指向新的尾节点




```
// 删除指定位置的节点
this.remove = function(index){
    // 参数不合法
    if(index < 0 || index >= length){
        return null;
    }else{
        var del_node = null;
        // 删除的是头节点
        if(index == 0){
            // head指向下一个节点
            del_node = head;
            head = head.next;
            // 如果head == null,说明之前链表只有一个节点
            if(!head){
                tail = null;
            }
```

```
            }else{
                // 找到索引为index-1的节点
                var pre_node = get_node(index-1);
                del_node = pre_node.next;
                pre_node.next = pre_node.next.next;
                // 如果删除的是尾节点
                if(del_node.next==null){
                    tail = pre_node;
                }
            }

            length -= 1;
            del_node.next = null;
            return del_node.data;
        }
    };
```

#### 2.2.5 get方法

获得指定索引位置的节点，需要传入参数index

```
    // 返回指定位置节点的值
    this.get = function(index){
        var node = get_node(index);
        if(node){
            return node.data;
        }
        return null;
    };
```

#### 2.2.6 indexOf方法

返回指定元素所在的位置，如果链表中没有这个元素，则返回-1，如果存在多个，则返回第一个，需要传入参数data。
使用一个while循环遍历链表，index表示元素索引位置，初始值为-1，curr_node指向头节点，while循环的条件是curr_node不为null，在循环体内，index加1，比较curr_node.data和data是否相等，如果相等，则返回index，反之curr_node指向下一个节点。
如果循环结束前仍然没有return，说明没要找到期望的data，返回-1

```
    // 返回指定元素的索引,如果没有,返回-1
    // 有多个相同元素,返回第一个
    this.indexOf = function(data){
```

```
        var index = -1;
        var curr_node = head;
        while(curr_node){
            index += 1
            if(curr_node.data == data){
                return index;
            }else{
                curr_node = curr_node.next;
            }
        }
        return -1;
    };
```

其他的方法较为简单，这里一并给出，完整代码如下

```
function LinkList(){
    // 定义节点
    var Node = function(data){
        this.data = data;
        this.next = null;
    }

    var length = 0;         // 长度
    var head = null;        // 头节点
    var tail = null;        // 尾节点

    // 添加一个新元素
    this.append = function(data){
        // 创建新节点
        var node = new Node(data);
        // 如果是空链表
        if(head==null){
            head = node;
            tail = head;
        }else{
            tail.next = node;       // 尾节点指向新创建的节点
            tail = node;            // tail指向链表的最后一个节点
        }
        length += 1;                // 长度加1
        return true;
    };

    // 返回链表大小
    this.length = function(){
        return length;
```

```
    };

    // 获得指定位置的节点
    var get_node = function(index){
        if(index < 0 || index >= length){
            return null;
        }
        var curr_node = head;
        var node_index = index;
        while(node_index-- > 0){
            curr_node = curr_node.next;
        }
        return curr_node;
    };

    // 在指定位置插入新的元素
    this.insert = function(index, data){
        // index == length,说明是在尾节点的后面新增,直接调用append方法即可
        if(index == length){
            return this.append(data);
        }else if(index > length || index < 0){
            // index范围错误
            return false;
        }else{
            var new_node = new Node(data);
            if(index == 0){
                // 如果在头节点前面插入,新的节点就变成了头节点
                new_node.next= head;
                head = new_node;
            }else{
                // 要插入的位置是index,找到索引为index-1的节点,然后进行连接
                var pre_node = get_node(index-1);
                new_node.next = pre_node.next;
                pre_node.next = new_node;
            }
            length += 1;
            return true;
        }
    };

    // 删除指定位置的节点
    this.remove = function(index){
        // 参数不合法
        if(index < 0 || index >= length){
            return null;
        }else{
```

```
            var del_node = null;
            // 删除的是头节点
            if(index == 0){
                // head指向下一个节点
                del_node = head;
                head = head.next;
                // 如果head == null,说明之前链表只有一个节点
                if(!head){
                    tail = null;
                }
            }else{
                // 找到索引为index-1的节点
                var pre_node = get_node(index-1);
                del_node = pre_node.next;
                pre_node.next = pre_node.next.next;
                // 如果删除的是尾节点
                if(del_node.next==null){
                    tail = pre_node;
                }
            }

            length -= 1;
            del_node.next = null;
            return del_node.data;
        }
    };

    // 删除尾节点
    this.remove_tail = function(){
        return this.remove(length-1);
    };

    // 删除头节点
    this.remove_head = function(){
        return this.remove(0);
    };

    // 返回指定位置节点的值
    this.get = function(index){
        var node = get_node(index);
        if(node){
            return node.data;
        }
        return null;
    };
```

```
// 返回链表头节点的值
this.head = function(){
    return this.get(0);
}

// 返回链表尾节点的值
this.tail = function(){
    return this.get(length-1);
}

// 返回指定元素的索引,如果没有,返回-1
// 有多个相同元素,返回第一个
this.indexOf = function(data){
    var index = -1;
    var curr_node = head;
    while(curr_node){
        index += 1
        if(curr_node.data == data){
            return index;
        }else{
            curr_node = curr_node.next;
        }
    }
    return -1;
};

// 输出链表
this.print = function(){
    var curr_node = head;
    var str_link = ""
    while(curr_node){

        str_link += curr_node.data.toString() + " ->";
        curr_node = curr_node.next;
    }
    str_link += "null";
    console.log(str_link);
    console.log("长度为"+ length.toString());
};

// isEmpty
this.isEmpty = function(){
    return length == 0;
};
```

```
    // 清空链表
    this.clear = function(){
        head = null;
        tail = null;
        length = 0;
    };

};


exports.LinkList = LinkList;
```

## 3、 基于链表实现的Stack 和 Queue

之前所学的Stack和Queue都是基于数组实现的，有了链表以后，则可以基于链表实现。

基于链表实现的栈：

```
function Stack(){
    var linklist = new LinkList();

    // 从栈顶添加元素
    this.push = function(item){
        linklist.append(item);
    };

    // 弹出栈顶元素
    this.pop = function(){
        return linklist.remove_tail();
    };

    // 返回栈顶元素
    this.top = function(){
        return linklist.tail();
    };

    // 返回栈的大小
    this.size = function(){
        return linklist.length();
    };

    // 判断是否为空
    this.isEmpty = function(){
        return linklist.isEmpty();
```

```
    };

    // 清空栈
    this.clear = function(){
        linklist.clear()
    };
};
```

基于链表实现的队列：

```
function Queue(){
    var linklist = new LinkList();

    // 入队列
    this.enqueue = function(item){
        linklist.append(item);
    };

    // 出队列
    this.dequeue = function(){
        return linklist.remove_head();
    };

    // 返回队首
    this.head = function(){
        return linklist.head();
    };

    // 返回队尾
    this.tail = function(){
        return linklist.tail();
    };

    // size
    this.size = function(){
        return linklist.length();
    };

    //clear
    this.clear = function(){
        linklist.clear();
    };

    // isEmpty
    this.isEmpty = function(){
```

```
        return linklist.isEmpty();
    };
};
```

## 4、链表常见面试题

### 4.1 翻转链表（困难模式）

#### 4.1.1 题目要求

使用迭代和递归两种方法翻转链表,下面的代码已经准备好了上下文环境，请实现函数reverse_iter和reverse_digui。

```
var Node = function(data){
    this.data = data;
    this.next = null;
}

var node1 = new Node(1);
var node2 = new Node(2);
var node3 = new Node(3);
var node4 = new Node(4);
var node5 = new Node(5);

node1.next = node2;
node2.next = node3;
node3.next = node4;
node4.next = node5;

function print(node){
    var curr_node = node;
    while(curr_node){
        console.log(curr_node.data);
        curr_node = curr_node.next;
    }
};

// 迭代翻转
function reverse_iter(head){
};
// 递归翻转
function reverse_digui(head){
}
```

#### 4.1.2 迭代翻转思路分析

在考虑算法时，多数情况下你考虑边界情况会让问题变得简单，但边界情况往往不具备普适性，因此，也要尝试考虑中间的情况，假设链表中间的某个点为curr_node，它的前一个节点是pre_node,后一个节点是next_node，现在把思路聚焦到这个curr_node节点上，只考虑在这一个点上进行翻转，翻转方法如下：

```
curr_node.next = pre_node;
```

只需要这简单的一个步骤就可以完成对curr_node节点的翻转,对于头节点来说，它没有上一个节点，让 pre_node=null,表示它的上一个节点是一个空节点。

在遍历的过程中，每完成一个节点的翻转，都让curr_node = next_node,找到下一个需要翻转的节点。同时，pre_node和next_node也跟随curr_node一起向后滑动。
示例代码：

```
// 迭代翻转
// 迭代翻转
function reverse_iter(head){
    if(!head){
        return null;
    }
    var pre_node = null;      // 前一个节点
    var curr_node = head;     // 当前要翻转的节点
    while(curr_node){
        var next_node = curr_node.next;     // 下一个节点
        curr_node.next = pre_node;          // 对当前节点进行翻转
        pre_node = curr_node;               // pre_node向后滑动
        curr_node = next_node;              // curr_node向后滑动
    }
    //最后要返回pre_node,当循环结束时,pre_node指向翻转前链表的最后一个节点
    return pre_node;
};
print(reverse_iter(node1));
```

#### 4.1.3 递归翻转链表思路分析

递归的思想，精髓之处在于甩锅，你做不到的事情，让别人去做，等别人做完了，你在别人的基础上继续做。

甩锅一共分为四步：

1. 明确函数的功能，既然是先让别人去做，那你得清楚的告诉他做什么。函数reverse_digui(head)完成的功能，是从head开始翻转链表，函数返回值是翻转后的头节点

2. 正式甩锅，进行递归调用，就翻转链表而言，甩锅的方法如下

   ```
   var new_head = reverse_digui(head.next);
   ```

   原本是翻转以head开头的链表，可是你不会啊，那就先让别人从head.next开始翻转链表，等他翻转完，得到的new_head就是翻转后的头节点。

3. 根据别人的结果，计算自己的结果
   第二步中，已经完成了从head.next开始翻转链表，现在，只需要把head连接到新链表上就可以了，新链表的尾节点是head.next，执行head.next.next = head，这样，head就成了新链表的尾节点。
   根据上面的三步，代码不难写出来

   ```
   // 递归翻转
   function reverse_digui(head){
   // 如果head 为null
   if(!head){
       return null;
   }
   // 从下一个节点开始进行翻转
   var new_head = reverse_digui(head.next);
   head.next.next = head;   // 把当前节点连接到新链表上
   head.next = null;
   return new_head;
   };
   ```

4. 找到递归的终止条件
   递归必须有终止条件，否则，就会进入到死循环，函数最终要返回新链表的头，而新链表的头正式旧链表的尾，所以，遇到尾节点时，直接返回尾节点，这就是递归终止的条件。

   ```
   // 递归翻转
   function reverse_digui(head){
   // 如果head 为null
   if(!head){
       return null;
   }
   if(head.next==null){
   ```

```
    return head;
}
// 从下一个节点开始进行翻转
var new_head = reverse_digui(head.next);
head.next.next = head;   // 把当前节点连接到新链表上
head.next = null;
return new_head;
};
print(reverse_digui(node1));
```

## 4.2 从尾到头打印链表（普通模式）

### 4.2.1 题目要求

从尾到头打印链表，不许翻转链表。

### 4.2.2 思路分析

当你拿到一个链表时，得到的是头节点，只有头结点以后的节点被打印了，才能打印头节点，这不正是一个可以甩锅的事情么,先定义函数

```
function reverse_print(head){

};
```

函数的功能，就是从head开始反向打印链表，但是现在不知道该如何反向打印，于是甩锅，先从head.next开始反向打印，等这部分打印完了，再打印head节点

### 4.2.3 示例代码

```
var Node = function(data){
    this.data = data;
    this.next = null;
}

var node1 = new Node(1);
var node2 = new Node(2);
var node3 = new Node(3);
var node4 = new Node(4);
var node5 = new Node(5);
```

```
node1.next = node2;
node2.next = node3;
node3.next = node4;
node4.next = node5;

function reverse_print(head){
    // 递归终止条件
    if(head==null){
        return
    }else{
        reverse_print(head.next);  // 甩锅
        console.log(head.data);    // 后面的都打印了，该打印自己了
    }
};

reverse_print(node1);
```

### 4.3 合并两个两个有序链表(困难模式)

#### 4.3.1 题目要求

已知有两个有序链表(链表元素从小到大)，请实现函数merge_link，将两个链表合并成一个有序链表，并返回新链表，原有的两个链表不要修改。

```
var Node = function(data){
    this.data = data;
    this.next = null;
}

var node1 = new Node(1);
var node2 = new Node(4);
var node3 = new Node(9);
var node4 = new Node(2);
var node5 = new Node(5);
var node6 = new Node(6);
var node7 = new Node(10);


node1.next = node2;
node2.next = node3;

node4.next = node5;
node5.next = node6;
```

```
node6.next = node7;

function merge_link(head1, head2){
    //在这里实现你的代码
};

print(merge_link(node1, node4));

function print(node){
    var curr_node = node;
    while(curr_node){
        console.log(curr_node.data);
        curr_node = curr_node.next;
    }
};
```

#### 4.3.2 思路分析

合并两个有序链表，是归并排序在链表上的一种实践。对两个链表，各自设置一个游标节点指向头节点，对游标节点上的数值进行比较，数值小的那个拿出来放入到合并链表中，同时游标节点向后滑动，继续比较游标节点数值大小。

为了实现滑动，需要使用一个while循环，当其中一个游标节点为null时，循环终止，这时，可能另一个游标节点还没有到达尾节点，那么把这段还没有遍历结束的链表添加到合并列表上。

#### 4.3.3 示例代码

```
function merge_link(head1, head2){
    if(head1 == null){
        return head2;
    }else if(head2 == null){
        return head1;
    }

    var merge_head = null;   // 合并后链表头
    var merge_tail = null;   // 合并后链表尾
    var curr_1 = head1;
    var curr_2 = head2;
    while(curr_1 && curr_2){
        // 找到最小值
        var min_data;
        if(curr_1.data < curr_2.data) {
```

```
            min_data = curr_1.data;
            curr_1 = curr_1.next;
        }else{
            min_data = curr_2.data;
            curr_2 = curr_2.next;
        }

        if(merge_head == null){
            merge_head = new Node(min_data);
            merge_tail = merge_head;
        }else{
            var new_node = new Node(min_data);
            // 把new_node连接到合并链表
            merge_tail.next = new_node;
            // 尾节点指向新创建的节点
            merge_tail = new_node;
        }

    }

    // 链表可能还有一部分没有合并进来
    var rest_link = null;
    if(curr_1){
        rest_link = curr_1;
    }
    if(curr_2){
        rest_link = curr_2;
    }

    while(rest_link){
        var new_node = new Node(rest_link.data);
        merge_tail.next = new_node;
        merge_tail = new_node;
        rest_link = rest_link.next;
    }
    return merge_head;
};
```

## 5. 课后作业

### 5.1 查找单链表中的倒数第K个节点（k > 0）（普通模式）

实现函数reverse_find，返回链表倒数第k个节点的数值

```
var Node = function(data){
    this.data = data;
    this.next = null;
}

var node1 = new Node(1);
var node2 = new Node(2);
var node3 = new Node(3);
var node4 = new Node(4);
var node5 = new Node(5);


node1.next = node2;
node2.next = node3;
node3.next = node4;
node4.next = node5;


function reverse_find(head, k){
    // 在这里实现你的代码,返回倒数第k个节点的值
    var fast = head;
    var slow = head;
    var step = k;
    // 先让快游标的先走k步
    while(step > 0 && fast){
        fast = fast.next;
        step -= 1;
    }

    // 当循环结束时,如果step != 0,说明链表的长度不够k
    if(step!=0){
        return null;
    }else{
        // 快的和慢的游标一起走
        while(fast && slow){
            fast = fast.next;
            slow = slow.next;
        }
    }
    return slow.data;
};

console.log(reverse_find(node1, 2));
```

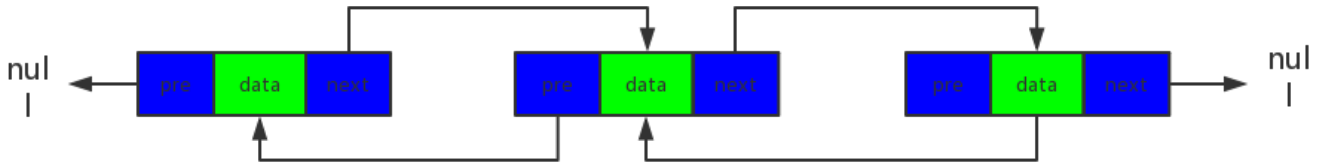
### 5.2 查找单链表的中间结点（普通模式）

实现函数find_middle，查找并返回链表的中间节点

```
var Node = function(data){
    this.data = data;
    this.next = null;
};

var node1 = new Node(1);
var node2 = new Node(2);
var node3 = new Node(3);
var node4 = new Node(4);
var node5 = new Node(5);


node1.next = node2;
node2.next = node3;
node3.next = node4;
node4.next = node5;


function find_middle(head){
    // 在这里实现你的代码,返回倒数第k个节点的值
    var fast = head;
    var slow = head;
    // 两个一起走,fast一次走两步,slow一次走一步
    while(fast.next){
        slow = slow.next;
        fast = fast.next.next;
    }

    return slow.data;
};

console.log(find_middle(node1));
```

### 5.3 实现双向链表（地狱模式）

下图为双向链表结构图




和单链表不同的是，双向链表的每个节点多出来一个pre指针域，指向它的前驱节点。
下面是双向链表的定义,请实现append，insert,remove这三个方法，其他的方法如果你能力足够强，也可以一并实现。

```
function DoubleLinkList() {
    // 定义节点
    var Node = function (data) {
        this.data = data;   // 数据
        this.next = null;   // 后继指针
        this.pre = null;    // 前驱指针
    }

    var length = 0;         // 长度
    var head = null;        // 头节点
    var tail = null;        // 尾节点

    this.append = function(data){
        // 在这里实现append方法
    };

    this.insert = function(index, data){
        // 在这里实现insert方法
    };

    this.remove = function(index){
        // 在这里实现remove方法
    };

};
```

## 数据结构之----BitMap

### 1、一个简单的问题

已知有n整个整数，这些整数的范围是[0, 100]，请你设计一种数据结构，使用数组存储这些数据，并提供两种方法，分别是addMember 和 isExist，下面是这种数据结构的类的定义，

请你实现它的两个方法

```
function FindClass(){
    var datas = [];    //存储数据

    // 加入一个整数 member
    this.addMember = function(member){

    };

    // 判断member是否存在
    this.isExist = function(member){

    };
}


var fc = new FindClass();
var arr = [0, 3, 5, 6, 9, 34, 23, 78, 99];
for(var i = 0;i < arr.length;i++){
    fc.addMember(arr[i]);
}

console.log(fc.isExist(3));
console.log(fc.isExist(7));
console.log(fc.isExist(78));
```

当你实现addMember和isExist之后，执行上面的代码，预期的输出结果是

```
true
false
true
```

### 1.1 毫无难度的实现

这种算法题，没有任何难度，只要会使用数组，就可以实现,先来看addMember的实现

```
    // 加入一个整数 member
    this.addMember = function(member){
        datas.push(member);
    };
```

接下来看isExist的实现，我这里提供两种方法
方法1

```
    // 判断member是否存在
    this.isExist = function(member){
        for(var i = 0;i < datas.length; i++){
            if(datas[i]==member){
                return true;
            }
        }
        return false;
    };
```

方法2

```
    // 判断member是否存在
    this.isExist = function(member){
        if(datas.indexOf(member) >= 0){
            return true;
        }
        return false;
    };
```

### 1.2 更快的方法

现在，题目难度升级，前面的实现虽然满足要求，但是速度慢，不论使用for循环查找，还是使用indexOf方法，时间复杂度都是O(n)，加入的元素越多，isExist方法的速度越慢，我们需要一个时间复杂度是O(1)的算法，不论向里面增加了多少数据，isExist的执行速度都是常量时间。

通过索引操作数据，时间复杂度就是O(1)，题目说明这些数在0到100之间，那么就用每个数自身的值作为索引，比如对于3这个数，可以让data[3] = 1，就表示把3添加进来了，data[2] = 0，表示2没有添加进来，那么这样一来，isExist方法就可以利用索引来判断member是否存在。
实现代码如下:

```
function FindClass(){
    var datas = new Array(100);     //存储数据
    // 先都初始化成0
    for(var i = 0;i < datas.length;i++){
        datas[i] = 0;
    }
```

```
    // 加入一个整数 member
    this.addMember = function(member){
        datas[member] = 1;
    };

    // 判断member是否存在
    this.isExist = function(member){
        if(datas[member] == 1){
            return true;
        }
        return false;
    };
}


var fc = new FindClass();
var arr = [0, 3, 5, 6, 9, 34, 23, 78, 99];
for(var i = 0;i < arr.length;i++){
    fc.addMember(arr[i]);
}

console.log(fc.isExist(3));
console.log(fc.isExist(7));
console.log(fc.isExist(78));
```

刚刚，我们实现了一个更加快速的算法，但这不是终点。

### 1.3 更节省空间的算法

1.2 中实现的算法，已经很快，但是却面临一个新的问题，如果数据非常多，多达1个亿，每个整数是4个字节，那么一亿个整数数就是4亿个字节，1024字节是1kb，1024kb是1M,4亿个字节就381M的内存空间。

如果没有这么多内存该怎么办？我们需要一种数据压缩的算法，用很少的空间来表示这一亿个数的存在与否。

## 2. 街边的路灯

街边有8栈路灯，编号分别是1 2 3 4 5 6 7 8 ，其中2号，5号，7号，8号路灯是亮着的，其余的都处于不亮的状态，请你设计一种简单的方法来表示这8栈路灯亮与不亮的状态。

一个非计算机专业的人看到这个问题，多半会嗤之以鼻，这算什么问题，答案是2578，表示这些编号的路灯是亮的，其余的都是不亮的。

而一个计算机专业的人看到这个问题，应该回答说75，因为75的二进制表示是0 1 0 0 1 0 1 1，恰好与路灯亮灭的状态相对应

```
1 2 3 4 5 6 7 8
0 1 0 0 1 0 1 1
```

仅仅是8个bit位就能表示8栈路灯的亮灭情况，那么一个整数有32个bit位，就可以表示32栈路灯的亮灭情况，回想我们在第一小节中遇到的问题，isExist方法要判断一个数是否存在，是不是也可以借助这种表达方式呢?

value是一个int类型的数据，初始化成0，当addMember传进来的参数是0的时候，就把value的二进制的第1位设置为1，二进制表示就是

```
00000000 00000000 00000000 00000001
```

此时，value = 1
如果又增加了一个3，就把value的二进制的第4为设置为1，二进制表示就是

```
00000000 00000000 00000000 00001001
```

此时，value = 9, 9可以表示 0 和 3都存在，一个整数，其实可以表示0~31的存在与否，如果我创建一个大小为10的数组，数组里存储整数，那么这个数组就可以表示0~319的存在与否

```
datas[0]  表示0~31存在与否
datas[1]  表示32~63存在与否
....
datas[9]  表示288~319存在与否
```

通过这种方式，就可以把空间的使用降低到原来的32分之一，存储一亿个整数的存在与否，只需要12M的内存空间,那么该如何对整数的二进制位进行操作呢。

## 3. 位运算

内存中的数据，最终的存储方式都是二进制，位运算就是对整数在内存中的二进制位进行操作。

### 3.1 按位与 &

两个整数进行按位与运算，相同二进制位的数字如果都是1，则结果为1，有一个位为0，则结

果为0， 下面是 3 & 7的计算过程

```
二进制     整数

0 1 1      3
1 1 1      7

0 1 1      3
```

3 & 7 = 3

### 3.2 按位或 |

两个整数进行按位或运算，相同二进制位的数字如果有一个为1，则结果为1，都为0，则结果为0， 下面是 5 | 8 的计算过程

```
二进制         整数

0 1 0 1        5
1 0 0 0        8

1 1 0 1        13
```

5 | 8 = 13

### 3.3 左移 <<

二进制向左移动 n 位，在后面添加 n 个0
下面是 3<<1 计算过程。

```
二进制         整数

    1 1        3

  1 1 0        6
```

3<<1 = 6

### 3.4 小试牛刀

一组数，内容以为 3， 9， 19， 20，请用一个整数来表示这些四个数

```
var value = 0;
value = value | 1<<3;
value = value | 1<<9;
value = value | 1<<19;
value = value | 1<<20;
console.log(value);
```

程序输出结果为1573384

## 4. bitmap

### 4.1 新的实现方式

经过前面一系列的分析和位运算学习，现在，我们要重新设计一个类，实现addMember和isExist方法，用更快的速度，更少的内存。

- 数据范围是0到100，那么只需要4个整数就可以表示4*32个数的存在与否，创建一个大小为4的数组
- 执行addMember时，先用member/32，确定member在数组里的索引（arr_index），然后用member%32，确定在整数的哪个二进制位行操作(bit_index)，最后执行bit_arr[arr_index] = bit_arr[arr_index] | 1<<bit_index;
- 执行isExist时，先用member/32，确定member在数组里的索引（arr_index），然后用member%32，确定在整数的哪个二进制位行操作(bit_index)，最后执行bit_arr[arr_index] & 1<<bit_index，如果结果不为0，就说明memeber存在

新的实现方法

```
function BitMap(size){
    var bit_arr = new Array(size);
    for(var i=0;i<bit_arr.length;i++){
        bit_arr[i] = 0;
    }

    this.addMember = function(member){
        var arr_index = Math.floor(member / 32);       // 决定在数组中的索引
        var bit_index = member % 32;                  // 决定在整数的32个bit位的哪一位
上
        bit_arr[arr_index] = bit_arr[arr_index] | 1<<bit_index;
    };

    this.isExist = function(member){
```

```
        var arr_index = Math.floor(member / 32);      // 决定在数组中的索引
        var bit_index = member % 32;                 // 决定在整数的32个bit位的哪一位
上
        var value = bit_arr[arr_index] & 1<<bit_index;
        if(value != 0){
            return true;
        }
        return false;
    };
}

var bit_map = new BitMap(4);
var arr = [0, 3, 5, 6, 9, 34, 23, 78, 99];
for(var i = 0;i < arr.length;i++){
    bit_map.addMember(arr[i]);
}

console.log(bit_map.isExist(3));
console.log(bit_map.isExist(7));
console.log(bit_map.isExist(78));
```

程序运行结果为

```
true
false
true
```

### 4.2 概念

不知不觉中，我们实现了一种数据结构，这种数据结构基于位做映射，能够用很少的内存存储数据，和数组不同，它只能存储表示某个数是否存在，可以用于大数据去重，大数据排序，两个集合取交集。

BitMap在处理大数据时才有优势，而且要求数据集紧凑，如果要处理的数只有3个，1，1000， 100000，那么空间利用率太低了，最大的值决定了BitMap要用多少内存。

### 4.3 大数据排序

有多达10亿无序整数，已知最大值为15亿，请对这个10亿个数进行排序。
传统排序算法都不可能解决这个排序问题，即便内存允许，其计算时间也是漫长的，如果使用BitMap就极为简单。

BitMap存储最大值为15亿的集合，只需要180M的空间，空间使用完全可以接受，至于速度，

存储和比较过程中的位运算速度都非常快，第一次遍历，将10亿个数都放入到BitMap中，第二次，从0到15亿进行遍历，如果在BitMap，则输出该数值，这样经过两次遍历，就可以将如此多的数据排序。

为了演示方便，只用一个很小的数组，[0, 6, 88, 7, 73, 34, 10, 99, 22]，已知数组最大值是99，利用BitMap排序的算法如下

```
var arr = [0, 6, 88, 7, 73, 34, 10, 99, 22];
var sort_arr = [];

var bit_map = new BitMap(4);
for(var i = 0;i < arr.length;i++){
    bit_map.addMember(arr[i]);
}

for(var i = 0;i <= 99;i++){
    if(bit_map.isExist(i)){
        sort_arr.push(i);
    }
}
console.log(sort_arr);
```

程序输出结果为

```
[ 0, 6, 7, 10, 22, 34, 73, 88, 99 ]
```

需要强调的一点，利用BitMap排序，待排序的集合中不能有重复数据。

## 5. 布隆过滤器

### 5.1 概念

前面所讲的BitMap的确很厉害，可是，却有着很强的局限性，BitMap只能用来处理整数，无法用于处理字符串，假设让你写一个强大的爬虫，每天爬取数以亿计的网页，那么你就需要一种数据结构，能够存储你已经爬取过的url，这样，才不至于重复爬取。

你可能会想到使用hash函数对url进行处理，转成整数，这样，似乎又可以使用BitMap了，但这样还是会有问题。假设BitMap能够映射的最大值是M，一个url的hash值需要对M求模，这样，就会产生冲突，而且随着存储数据的增多，冲突率会越来越大。

布隆过滤器的思想非常简单，其基本思路和BitMap是一样的，可以把布隆过滤器看做是BitMap的扩展。为了解决冲突率，布隆过滤器要求使用k个hash函数，新增一个key时，把

key散列成k个整数，然后在数组中将这个k个整数所对应的二进制位设置为1，判断某个key是否存在时，还是使用k个hash函数对key进行散列，得到k个整数，如果这k个整数所对应的二进制位都是1，就说明这个key存在，否则，这个key不存在。

对于一个布隆过滤器，有两个参数需要设置，一个是预估的最多存放的数据的数量，一个是可以接受的冲突率。

假设预估最多存放n个数据，可已接受的冲突率是p，那么就可以计算出来布隆过滤器所需要的bit位数量m,也可以计算所需要的hash函数的个数k,计算公式如下

$$m = -\frac{n*ln^p}{(ln^2)^2}$$

$$k = ln^2 * \frac{m}{n}$$

这两个公式，你只需做到知道即可，如果你数学功底非常好，也可以自己来推导，这里有一篇文章，你可以用来参考[布隆过滤器参数计算]

### 5.2 hash函数

哈希函数就是将某一不定长的对象映射为另一个定长的对象，如果你对这个概念感到困惑，你就换一个理解方法，你给hash函数传入一个字符串，它返回一个整数。为了实现一个布隆过滤器，我们需要一个好的hash函数，计算快，冲突又少，很幸运，有很多开源的hash算法，我在github上找到了一个murmurhash的实现，代码如下

```
function murmurhash3_32_gc(key, seed) {
    var remainder, bytes, h1, h1b, c1, c1b, c2, c2b, k1, i;

    remainder = key.length & 3; // key.length % 4
    bytes = key.length - remainder;
    h1 = seed;
    c1 = 0xcc9e2d51;
    c2 = 0x1b873593;
    i = 0;

    while (i < bytes) {
        k1 =
            ((key.charCodeAt(i) & 0xff)) |
            ((key.charCodeAt(++i) & 0xff) << 8) |
            ((key.charCodeAt(++i) & 0xff) << 16) |
            ((key.charCodeAt(++i) & 0xff) << 24);
        ++i;

        k1 = ((((k1 & 0xffff) * c1) + ((((k1 >>> 16) * c1) & 0xffff) << 16))) & 0x
ffffffff;
        k1 = (k1 << 15) | (k1 >>> 17);
```

```
        k1 = ((((k1 & 0xffff) * c2) + ((((k1 >>> 16) * c2) & 0xffff) << 16))) & 0x
ffffffff;

        h1 ^= k1;
        h1 = (h1 << 13) | (h1 >>> 19);
        h1b = ((((h1 & 0xffff) * 5) + ((((h1 >>> 16) * 5) & 0xffff) << 16))) & 0xf
fffffff;
        h1 = (((h1b & 0xffff) + 0x6b64) + ((((h1b >>> 16) + 0xe654) & 0xffff) << 1
6));
    }

    k1 = 0;

    switch (remainder) {
        case 3: k1 ^= (key.charCodeAt(i + 2) & 0xff) << 16;
        case 2: k1 ^= (key.charCodeAt(i + 1) & 0xff) << 8;
        case 1: k1 ^= (key.charCodeAt(i) & 0xff);

            k1 = (((k1 & 0xffff) * c1) + ((((k1 >>> 16) * c1) & 0xffff) << 16)) &
0xffffffff;
            k1 = (k1 << 15) | (k1 >>> 17);
            k1 = (((k1 & 0xffff) * c2) + ((((k1 >>> 16) * c2) & 0xffff) << 16)) &
0xffffffff;
            h1 ^= k1;
    }

    h1 ^= key.length;

    h1 ^= h1 >>> 16;
    h1 = (((h1 & 0xffff) * 0x85ebca6b) + ((((h1 >>> 16) * 0x85ebca6b) & 0xffff) <<
 16)) & 0xffffffff;
    h1 ^= h1 >>> 13;
    h1 = ((((h1 & 0xffff) * 0xc2b2ae35) + ((((h1 >>> 16) * 0xc2b2ae35) & 0xffff) <
< 16))) & 0xffffffff;
    h1 ^= h1 >>> 16;

    return h1 >>> 0;
}
```

murmurhash3_32_gc函数接受两个值，key是需要散列的对象，seed是种子，同一个key，不同的种子会返回不同的结果，这正是我们想要的，布隆过滤器需要k个hash值，对于一个key，传入k个种子就可以得到k个散列结果了。

### 5.3 BoolmFilter 类

根据前面对布隆过滤器的讲述，我定义一个BoolmFilter类

```
function BoolmFilter (max_count, error_rate) {
    // 位图映射变量
    var bitMap = [];
    // 最多可放的数量
    var max_count = max_count;
    // 错误率
    var error_rate = error_rate;
    // 位图变量的长度
    var bit_size = Math.ceil(max_count * (-Math.log(error_rate) / (Math.log(2) * M
ath.log(2)) ));
    // 哈希数量
    var hash_ount = Math.ceil(Math.log(2) * (bit_size / max_count));
};
```

我在实现某个具体类时，都会先把属性和构造类对象的参数定义清楚。
接下来要实现具体的方法，add和isExist。

### 5.3.1 操作二进制位

每次add的时候，都要把key散列成k个值，并将这个k个值对应的二进制位设置为1，那么设置为1的这个动作就需要执行k次，这种需要重复执行的操作，我们就应该单独作为一个函数来实现。

```
    // 设置位的值
    var set_bit = function (bit) {
        var arr_index = Math.floor(bit / 32);
        var bit_index = Math.floor(bit % 32);
        bitMap[arr_index] |= (1 << bit_index);
    }

    // 读取位的值
    var get_bit = function (bit) {
        var arr_index = Math.floor(bit / 32);
        var bit_index = Math.floor(bit % 32);
        return bitMap[arr_index] &= (1 << bit_index);
    }
```

### 5.3.2 add方法

```
    // 添加key
    this.add = function (key) {
        if (this.isExist(key)) {
            return -1;  //表示已经存在
        }

        for (var i = 0; i < hash_ount; i++) {
            var hash_value = murmurhash3_32_gc(key, i);
            set_bit(Math.abs(Math.floor(hash_value % (bit_size))));
        }
    }
```

### 5.3.3 isExist方法

```
    // 检测是否存在
    this.isExist = function (key) {
        for (var i = 0; i < hash_ount; i++) {
            var hash_value = murmurhash3_32_gc(key, i);
            if (!get_bit(Math.abs(Math.floor(hash_value % (bit_size))))) {
                return false;
            }
        }

        return true;
    };
```

最终完整的实现代码

```
function murmurhash3_32_gc(key, seed) {
    var remainder, bytes, h1, h1b, c1, c1b, c2, c2b, k1, i;

    remainder = key.length & 3; // key.length % 4
    bytes = key.length - remainder;
    h1 = seed;
    c1 = 0xcc9e2d51;
    c2 = 0x1b873593;
    i = 0;

    while (i < bytes) {
        k1 =
            ((key.charCodeAt(i) & 0xff)) |
            ((key.charCodeAt(++i) & 0xff) << 8) |
```

```
            ((key.charCodeAt(++i) & 0xff) << 16) |
            ((key.charCodeAt(++i) & 0xff) << 24);
        ++i;

        k1 = ((((k1 & 0xffff) * c1) + ((((k1 >>> 16) * c1) & 0xffff) << 16))) & 0x
ffffffff;
        k1 = (k1 << 15) | (k1 >>> 17);
        k1 = ((((k1 & 0xffff) * c2) + ((((k1 >>> 16) * c2) & 0xffff) << 16))) & 0x
ffffffff;

        h1 ^= k1;
        h1 = (h1 << 13) | (h1 >>> 19);
        h1b = ((((h1 & 0xffff) * 5) + ((((h1 >>> 16) * 5) & 0xffff) << 16))) & 0xf
fffffff;
        h1 = (((h1b & 0xffff) + 0x6b64) + ((((h1b >>> 16) + 0xe654) & 0xffff) << 1
6));
    }

    k1 = 0;

    switch (remainder) {
        case 3: k1 ^= (key.charCodeAt(i + 2) & 0xff) << 16;
        case 2: k1 ^= (key.charCodeAt(i + 1) & 0xff) << 8;
        case 1: k1 ^= (key.charCodeAt(i) & 0xff);

            k1 = (((k1 & 0xffff) * c1) + ((((k1 >>> 16) * c1) & 0xffff) << 16)) &
0xffffffff;
            k1 = (k1 << 15) | (k1 >>> 17);
            k1 = (((k1 & 0xffff) * c2) + ((((k1 >>> 16) * c2) & 0xffff) << 16)) &
0xffffffff;
            h1 ^= k1;
    }

    h1 ^= key.length;

    h1 ^= h1 >>> 16;
    h1 = (((h1 & 0xffff) * 0x85ebca6b) + ((((h1 >>> 16) * 0x85ebca6b) & 0xffff) <<
 16)) & 0xffffffff;
    h1 ^= h1 >>> 13;
    h1 = ((((h1 & 0xffff) * 0xc2b2ae35) + ((((h1 >>> 16) * 0xc2b2ae35) & 0xffff) <
< 16))) & 0xffffffff;
    h1 ^= h1 >>> 16;

    return h1 >>> 0;
}
```

```
function BoolmFilter (max_count, error_rate) {
    // 位图映射变量
    var bitMap = [];
    // 最多可放的数量
    var max_count = max_count;
    // 错误率
    var error_rate = error_rate;
    // 位图变量的长度
    var bit_size = Math.ceil(max_count * (-Math.log(error_rate) / (Math.log(2) * M
ath.log(2)) ));
    // 哈希数量
    var hash_ount = Math.ceil(Math.log(2) * (bit_size / max_count));


    // 设置位的值
    var set_bit = function (bit) {
        var arr_index = Math.floor(bit / 31);
        var bit_index = Math.floor(bit % 31);
        bitMap[arr_index] |= (1 << bit_index);
    };

    // 读取位的值
    var get_bit = function (bit) {
        var arr_index = Math.floor(bit / 31);
        var bit_index = Math.floor(bit % 31);
        return bitMap[arr_index] &= (1 << bit_index);
    };

    // 添加key
    this.add = function (key) {
        if (this.isExist(key)) {
            return -1;  //表示已经存在
        }

        for (var i = 0; i < hash_ount; i++) {
            var hash_value = murmurhash3_32_gc(key, i);
            set_bit(Math.abs(Math.floor(hash_value % (bit_size))));
        }
    };

    // 检测是否存在
    this.isExist = function (key) {
        for (var i = 0; i < hash_ount; i++) {
            var hash_value = murmurhash3_32_gc(key, i);
```

```
            if (!get_bit(Math.abs(Math.floor(hash_value % (bit_size))))) {
                return false;
            }
        }

        return true;
    };
};

var bloomFilter = new BoolmFilter(1000000, 0.01);

bloomFilter.add('https://blog.csdn.net/houzuoxin/article/details/20907911');
bloomFilter.add('https://www.jianshu.com/p/888c5eaebabd');
console.log(bloomFilter.isExist('https://blog.csdn.net/houzuoxin/article/details/2
0907911'));
console.log(bloomFilter.isExist('https://www.jianshu.com/p/888c5eaebabd'));
console.log(bloomFilter.isExist('https://www.jianshu.com/p/888c5eaebabd435'));
```

### 6.课后练习题

#### 6.1 两个集合取交集（普通模式）

两个数组，内容分别为[1, 4, 6, 8, 9, 10, 15], [6, 14, 9, 2, 0, 7]，请用BitMap计算他们的交集

#### 6.2 支持负数（困难模式）

课程里所讲的BitMap只能存储大于等于0的整数，请你改造BitMap类，不论正数还是负数，都可以存储并判断是否存在

#### 6.3 查找不重复的数（地狱模式）

有一个数组，内容为[1, 3, 4, 5, 7, 4, 8, 9, 2, 9]，有些数值是重复出现的，请你改造BitMap类，增加一个isRepeat(member)方法，判断member是否重复出现，并利用该方法找出数组中不重复的数。

## 数据结构之----树

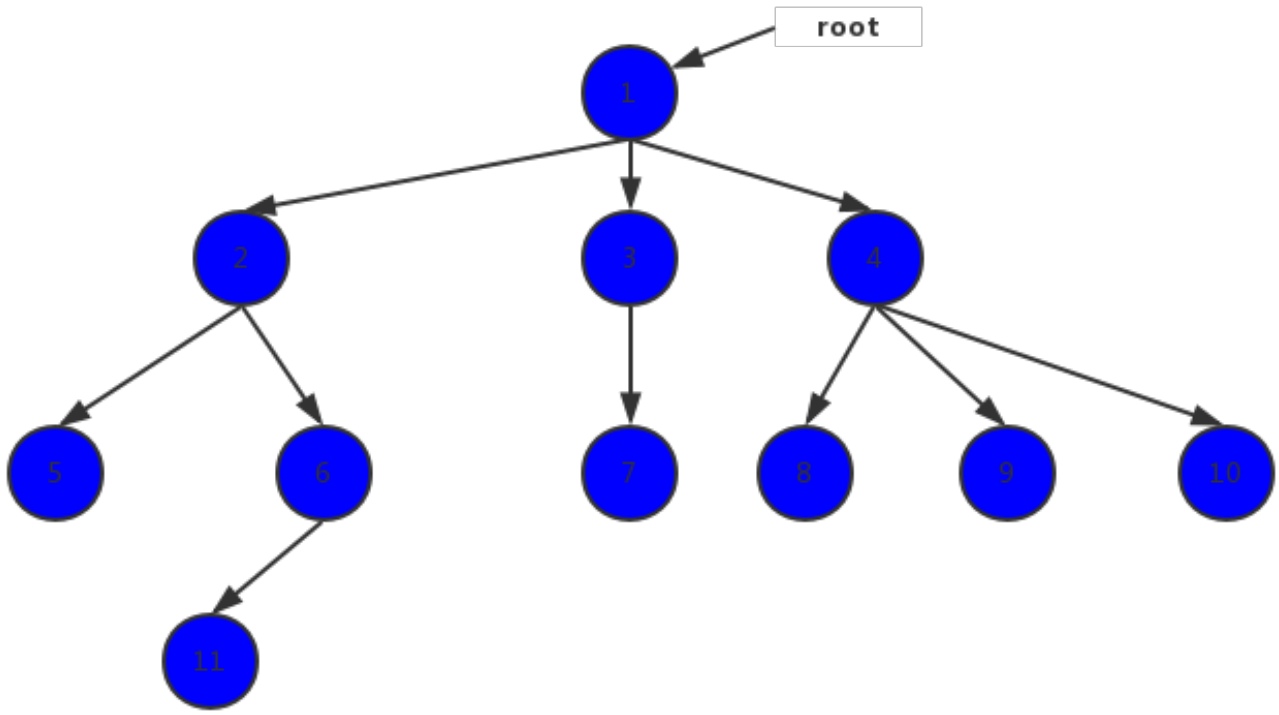
### 1. 概念介绍

#### 1.1 结构

树是一种非线性的数据结构，是由n(n>=0)个节点组成的集合。

- 如果n=0,是一颗空数
- 如果n>0,树有一个特殊的节点，这个节点没有父节点，被称为根节点（root）
- 除根结点之外的其余数据元素被分为m（m≥0）个互不相交的集合T1，T2，……Tm−1，其中每一个集合Ti（1<=i<=m）本身也是一棵树，被称作原树的子树

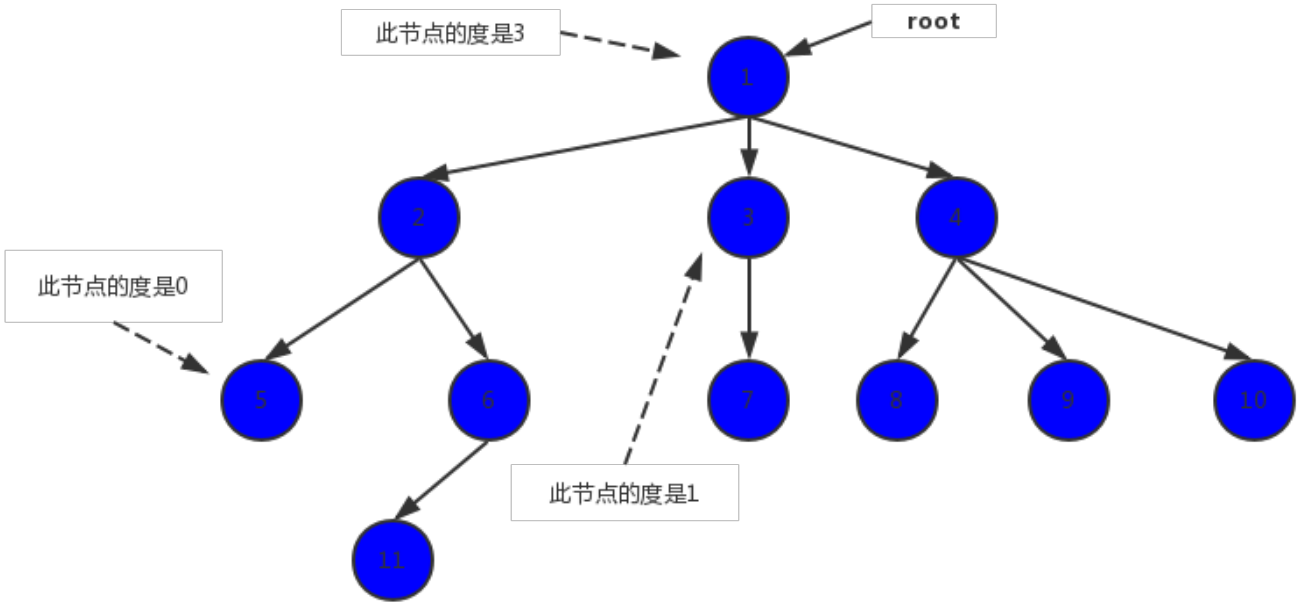
下图是一颗数的结构图




### 1.2 节点

它包含数据项，和指向其他节点的指针，上图中的树，有7个节点

### 1.3 节点的度




### 1.4 叶节点

度为0的节点被成为叶节点， 如上图中的 5 6 7 8 9 10

### 1.5 分支节点

除叶节点外的节点就是分支节点， 如上图中的 1 2 3 4

### 1.5 子女节点

若节点x有子树，则这颗子树的根节点就是节点x的子女节点，例如2 3 4 都是1的子女节点

### 1.6 父节点

若节点x有子女节点，则x为子女节点的父节点，例如 1是 2 3 4 的父节点，2 是 5 6 的父节点

### 1.7 兄弟节点

同一个父节点的子女节点互称为兄弟， 如 5 和 6 是兄弟节点

### 1.8 祖先节点

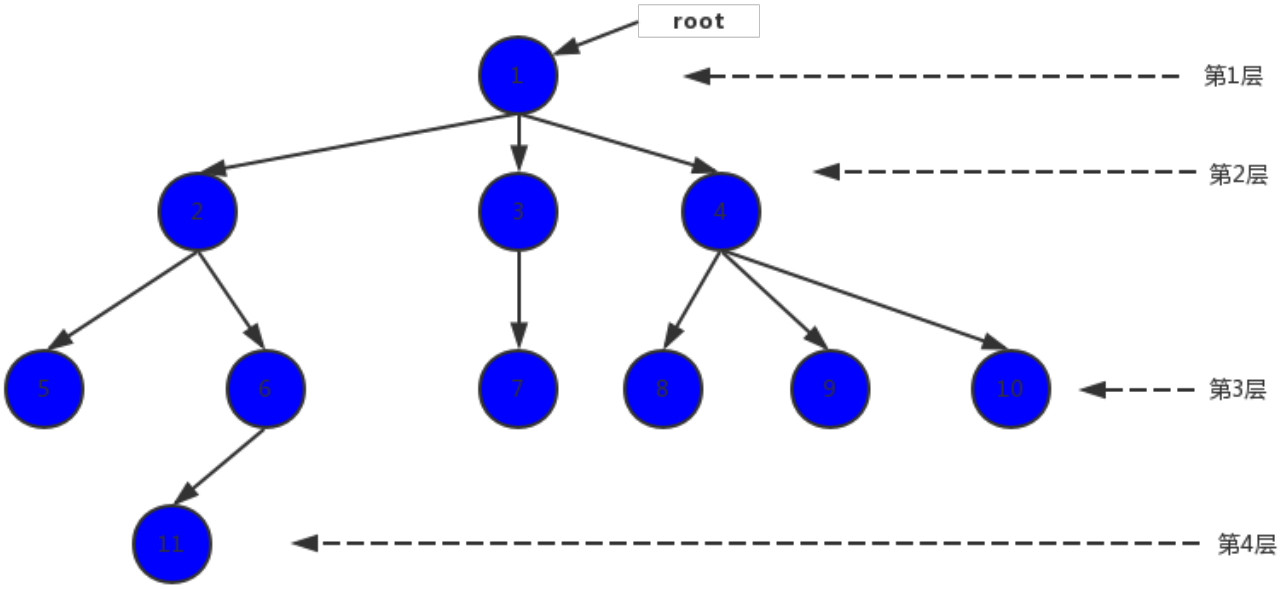
从根节点到该节点所进过分支上的所有节点，如节点5 ， 它的祖先节点为 1 2

### 1.9 子孙节点

某一个节点的子女，以及这些子女的子女都是该节点的子孙节点，如节点2 ，5 6 11 都是它的子孙节点

### 1.10 节点所在层次

根节点在第一层，它的子女在第二层，以此类推




### 1.11 树的深度

树中距离根节点最远的节点所处的层次就是树的深度，上图中，树的深度是4

### 1.12 树的高度

叶节点的高度为1，非叶节点的高度是它的子女节点高度的最大值加1，高度与深度数值相等，但计算方式不一样，上图中树的

### 1.13 树的度

树中节点的度的最大值，上图中，所有节点的度的最大值是3，树的度就是3

### 1.14 有序树

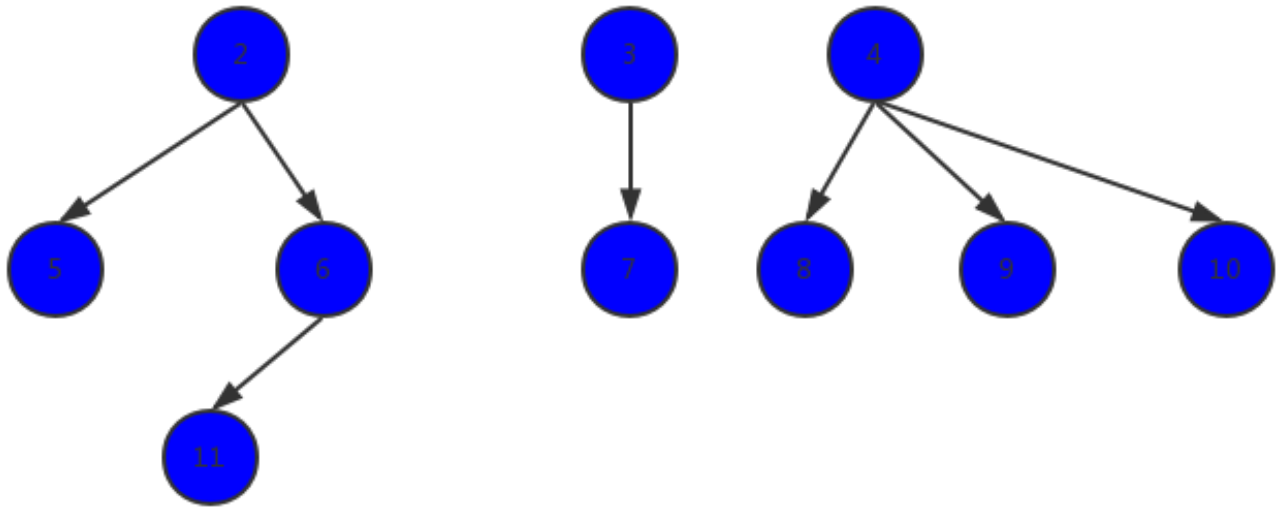
树中节点的各棵字树 T1，T2...是有次序的，T1是第1棵子树，T2是第2棵子树

### 1.15 无序树

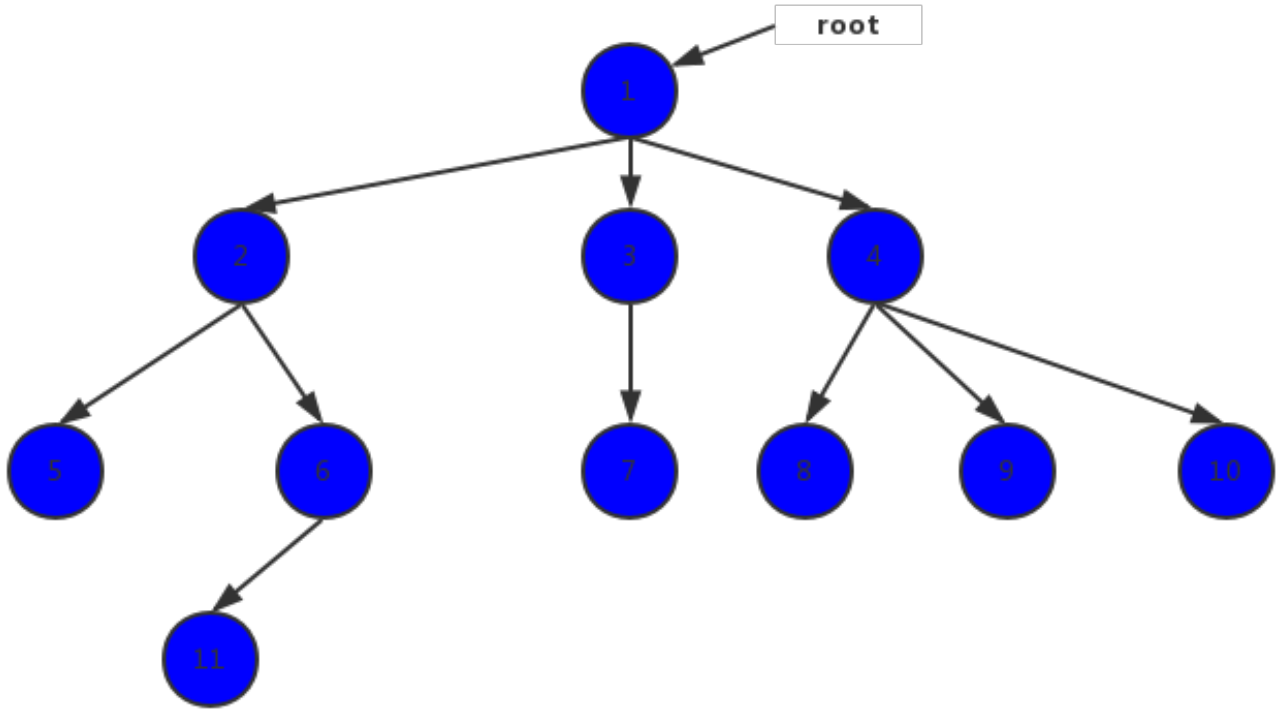
树中节点的各棵子树之间的次序不重要，可以互相交换位置

### 1.16 森林

森林是m(m>=0)棵树的集合



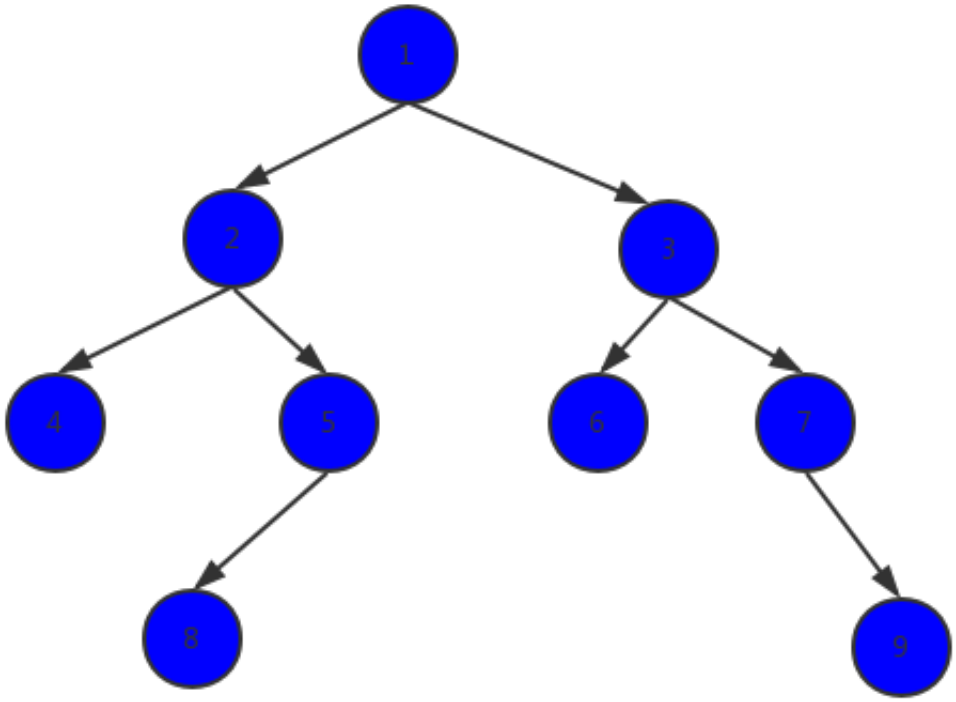

这片森林里有3棵树，如果我们增加一个根节点1，让2 3 4 成为它的子女，那么就森林就变成了一颗树



删掉根节点，就变成了森林。

## 2. 二叉树

二叉树是树的一种特殊情况，每个节点最多有有两个子女，分别称为该节点的左子女和右子女，就是说，在二叉树中，不存在度大于2的节点。二叉树的子树有左右之分，次序不能颠倒，下图是一个二叉树的结构图



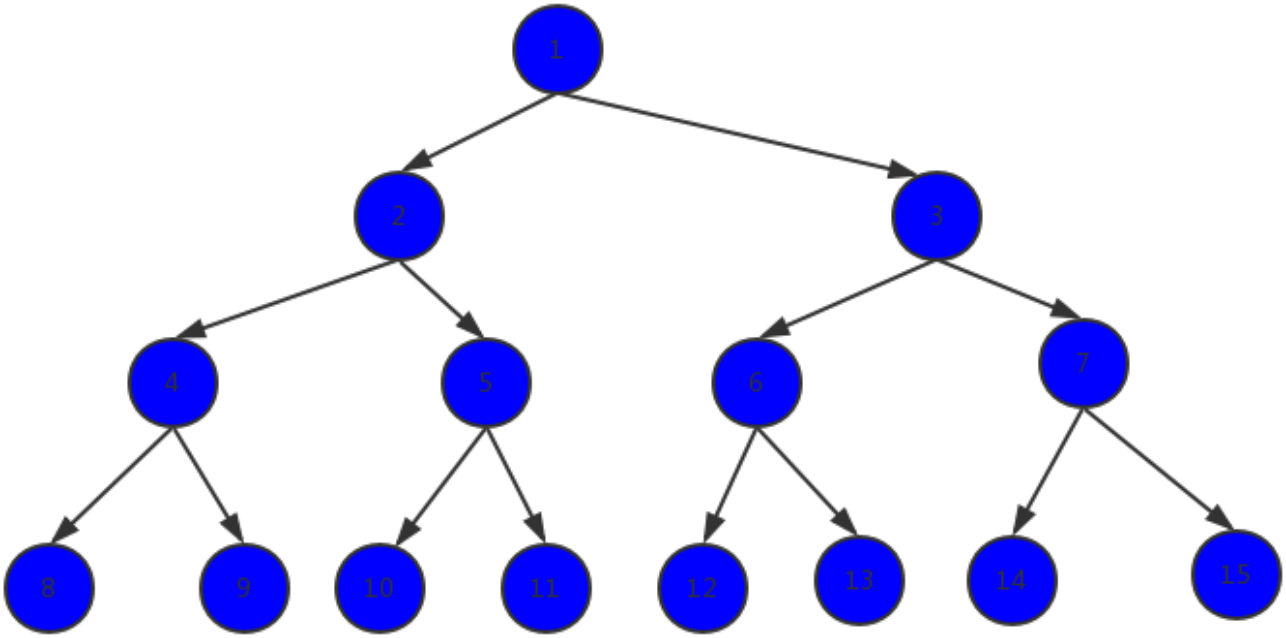

### 2.1 二叉树的性质

- 在二叉树的第i(i>=1)层，最多有$2^{i-1}$ 个节点
- 深度为k(k>=0)的二叉树，最少有k个节点，最多有$2^{k-1}$ 个节点
- 对于一棵非空二叉树，叶节点的数量等于度为2的节点数量加1

### 2.2 特殊二叉树

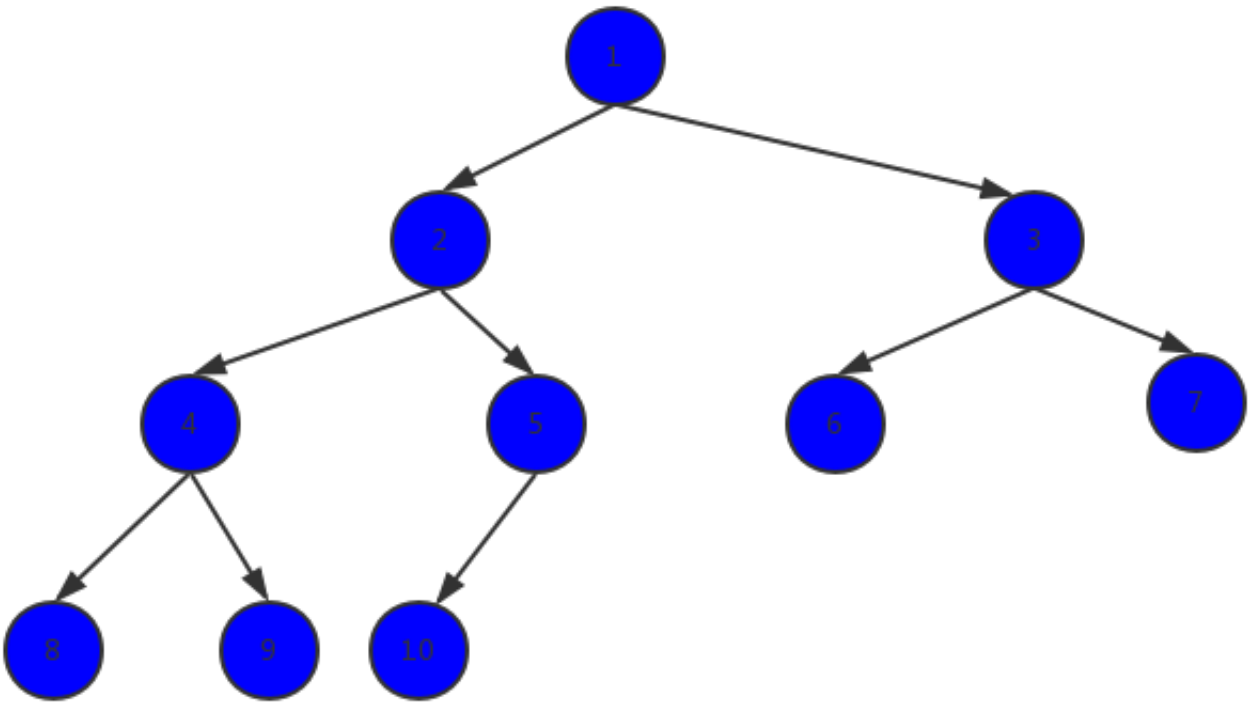
#### 2.2.1 满二叉树

深度为k的满二叉树，是有$2^{k-1}$ 个节点的二叉树，每一层都达到了可以容纳的最大数量的节点




### 2.2.2 完全二叉树

深度为k的完全二叉树，从第1层到第k–1层都是满的，第k层，或是满的或是从右向左连续缺若干个节点



### 2.3 二叉树的类定义

#### 2.3.1 定义节点

```
var BinTreeNode = function(data){
    this.data = data;
    this.leftChild = null;     // 左孩子
    this.rightChild = null;    // 右孩子
    this.parentNode = null;    // 父节点
};
```

data 仍然表示数据域，其余三个都是指针域，指向其他节点

#### 2.3.2 定义二叉树类

```
function BinaryTree(){
    var root = null;   //根节点

    // 采用广义表表示的建立二叉树方法
    this.init_tree = function(string){

    }
};
```

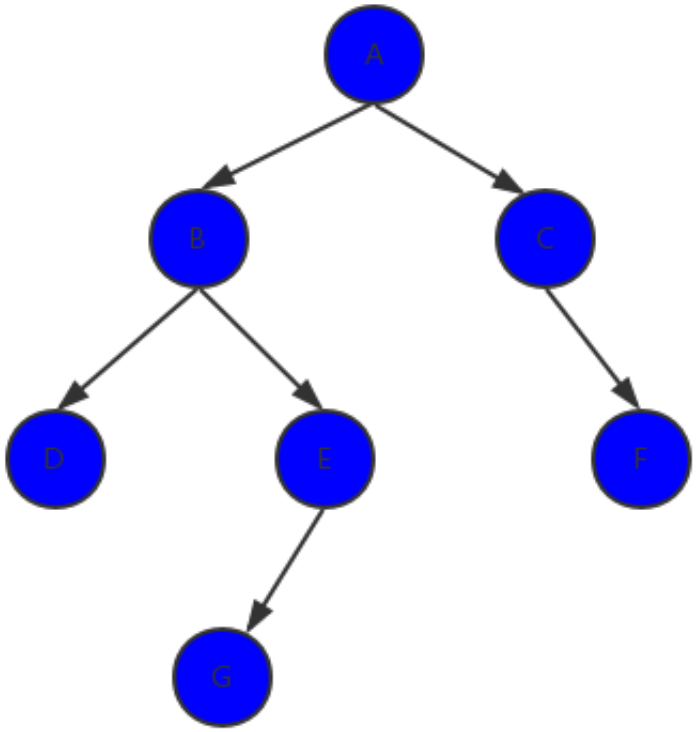
#### 2.3.3 init_tree

这里我们先实现init_tree,它接收一个表示二叉树的广义表，创建一颗树。

如何用广义表来表达二叉树呢，以广义表 A(B(D,E(G,)),C(,F))# 为例，算法如下

- 广义表的表名放在表前，表示树的根节点，括号中的是根的左右子树
- 每个节点的左右子树用逗号隔开，如果有仅有右子树没有左子树，逗号不省略
- 整个广义表的最后加上特殊符号#表示输入结束

广义表 A(B(D,E(G,)),C(,F))# 所代表的树结构如下图




遍历这个A(B(D,E(G,)),C(,F))# 字符串，来建立一颗二叉树。

遇到左括号的时候，说明前面有一个节点，这个括号里的两个节点都是它的子节点，但是子节点后面还会有子节点，因此，我们需要一个先进后出的数据结构，把前面的节点保存下来，这样，栈顶就是当前要处理的两个节点的父节点。

逗号分隔了左右子树，因此需要一个变量来标识遇到的是左子树还是右子树，假设这个变量为k，遇到左括号的时候，k=1，表示开始识别左子树，遇到逗号，k=2表示开始识别右子树。

遇到右括号，说明一棵子树结束了，那么栈顶的元素正是这棵子树的根节点，执行pop方法出栈。

```
    // 采用广义表表示的建立二叉树方法
    this.init_tree = function(string){
        var stack = new Stack.Stack();
        var k = 0;
        var new_node = null;
        for(var i =0; i < string.length;i++){
            var item = string[i];
            if(item=="("){
                stack.push(new_node);
                k = 1;
            }else if(item==")"){
                stack.pop();
```

```
            }else if(item==","){
                k = 2;
            }else{
                new_node = BinTreeNode(item);
                if(root==null){
                    root = new_node;
                }else if(k==1){
                    // 左子树
                    var top_item = stack.top();
                    top_item.leftChild = new_node;
                    new_node.parentNode = top_item;
                }else{
                    // 右子树
                    var top_item = stack.top();
                    top_item.rightChild = new_n
                    ode;
                    new_node.parentNode = top_item;
                }
            }
        }
    };
```

### 2.3.3 in_order 中序遍历算法

中序遍历，前序遍历，后序遍历，都可以借助递归轻松解决，递归的精髓在于甩锅
中序遍历，是先遍历节点的左子树，然后是当前节点，最后遍历节点的右子树

```
    // 中序遍历
    this.in_order = function(node){
        if(node==null){
            return;
        }
        this.in_order(node.leftChild);
        console.log(node.data);
        this.in_order(node.rightChild);
    };
```

### 2.3.4 pre_order 前序遍历算法

```
// 前序遍历
    this.pre_order = function(node){
        if(node==null){
```

```
            return;
        }
        console.log(node.data);
        this.pre_order(node.leftChild);
        this.pre_order(node.rightChild);
    };
```

### 2.3.5 pre_order 前序遍历算法

```
    // 后序遍历
    this.post_order = function(node){
        if(node==null){
            return;
        }
        this.post_order(node.leftChild);
        this.post_order(node.rightChild);
        console.log(node.data);
    };
```

### 2.3.6 size 返回节点数量

```
    var tree_node_count = function(node){
        // 左子树的节点数量加上右子树的节点数量 再加上1
        if(!node){
            return 0;
        }
        var left_node_count = tree_node_count(node.leftChild);
        var right_node_count = tree_node_count(node.rightChild);
        return left_node_count + right_node_count + 1;
    };
    // 返回节点数量
    this.size = function(){
        return tree_node_count(root);
    };
```

### 2.3.7 height 返回树的高度

```
    var tree_height = function(node){
        // 左子树的高度和右子树的高度取最大值,加上当前的高度
        if(!node){
            return 0;
```

```
        }

        var left_child_height = tree_height(node.leftChild);
        var right_child_height = tree_height(node.rightChild);
        if(left_child_height > right_child_height){
            return left_child_height + 1;
        }else{
            return right_child_height + 1;
        }

    };
    // 返回高度
    this.height = function(){
        return tree_height(root);
    };
```

### 2.3.8 查找节点

```
    var find_node = function(node, data){
        if(!node){
            return null;
        }
        if(node.data == data){
            return node;
        }

        left_res = find_node(node.leftChild, data);
        if(left_res){
            return left_res;
        }

        return find_node(node.rightChild, data);
    };
    // 查找data
    this.find = function(data){
        return find_node(root, data);
    };
```

## 2.4 课后练习题

### 2.4.1 求一棵树的镜像(普通模式)

对于一棵树，如果每个节点的左右子树互换位置，那么就变成了这棵树的镜像

请实现mirror方法

```
var bt = new BinaryTree();
bt.init_tree("A(B(D,E(G,)),C(,F))#");
var root_node = bt.get_root();

var mirror = function(node){
    // 在这里实现你的方法
};

mirror(root_node);
bt.in_order(root_node);
```

程序最终输出结果应该是

```
F C A E G  B D
```

### 2.4.2 使用非递归方式实现 前，中，后三种遍历方法（普通模式+）

### 2.4.3 寻找两个节点的最近公共祖先（困难模式）

```
var bt = new BinaryTree();
bt.init_tree("A(B(D,E(G,)),C(,F))#");
var root_node = bt.get_root();

var node1 = bt.find("D");
var node2 = bt.find("G");
console.log(node1.data);
console.log(node2.data);

// 寻找最近公共祖先
var lowest_common_ancestor = function(root_node, node1, node2){
    // 实现你的算法
};

var ancestor = lowest_common_ancestor(root_node, node1, node2);
console.log(ancestor.data);
```

程序最终输出结果为 B

### 2.4.4 分层打印二叉树（困难模式+）

```
var bt = new BinaryTree();
bt.init_tree("A(B(D,E(G,)),C(,F))#");
var root_node = bt.get_root();

// 层次遍历
var level_order = function(node){
    // 实现你的方法
};

level_order(root_node);
```

实现函数level_order ， 程序最终输出结果为

```
A
B  C
D  E  F
G
```

### 2.4.5 输出指定层的节点个数（困难模式+）

实现函数get_width 返回第n层的节点个数

```
var bt = new BinaryTree();
bt.init_tree("A(B(D,E(G,)),C(,F))#");
var root_node = bt.get_root();

// 获得宽度
var get_width = function(node, n){
    // 实现你的函数
};

console.log(get_width(root_node, 1));
console.log(get_width(root_node, 2));
console.log(get_width(root_node, 3));
console.log(get_width(root_node, 4));
```

程序输出结果为

```
1
2
3
```

```
1
```