基础篇一共七个小节，我按照重要程度将其划分为四个等级：

- T1：栈（队列），模拟与枚举
- T2：树，双指针
- T3：哈希表，链表
- T4：图（图的搜索后面会单独讲）

大家也可以针对自己面试的公司酌情修改优先级。 比如 google 喜欢考图，你就优先看图

本文我们讲解线性结构，而且是最简单的线性结构 - 《数组，栈，队列》，另外一种线性结构是第二节的《链表》。

我们先讲线性数据结构：数组和链表。后面再讲解更为复杂的非线性数据结构。非线性数据结构是基于线性数据结构的，大家一定先打好基础。

先导篇讲了数据结构就是如何操作一堆数据，而且这些数据是有关系的。那么如果集合中满足以下条件，这个数据结构就是线性的：

1. 集合只有一个首元素
2. 集合只有一个尾元素
3. 集合中的其他元素（除了首元素和尾元素）均有且仅有一个前驱和后继。

基于这个定义，在学习后面的内容的时候你就会明白为什么数据和链表是线性数据结构，而后面讲的二叉树，图等是非线性数据结构。

数组，栈，队列

大家好，本节是数据结构的开篇内容。本节主要讲述数组，栈以及队列。

> 数组的知识大家可以轻松地迁移到字符串，因此本书不对字符串进行特殊讲解。

数组

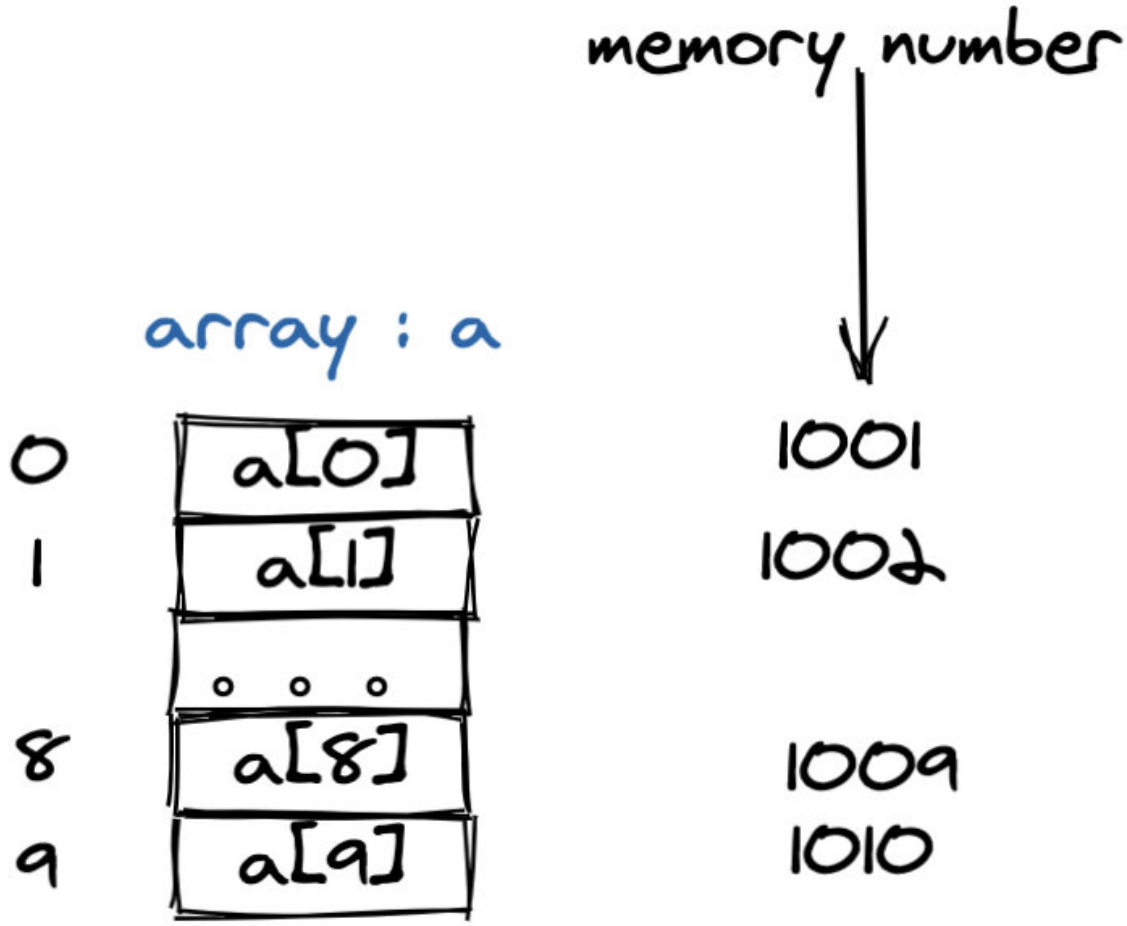
数组是一种使用最为广泛的数据结构，尤其是在大家的日常开发中，原因无非就是**操作简单**和**支持随机访问**。而字符串大家也可以将其看成是一个字符数组，这更加夯实了数组的重要性。

数组是我们要讲的第一个重要数据结构（另外一个是链表），很多的数据结构都是基于其产生的。比如后文要讲的二叉树，图等。

比如给你一个数组 parents，其中 parents[i]表示 i 的父节点。比如 [-1,0,0] 就表示 索引 1 和 2 的父节点是 0，而 0 没有任何的父节点，我们不妨用 -1 表示。这就完成了用数组来表示**二叉树**的过程。图也是类似的，后文我们讲图的时候，也会用数组来存图中边的关系。

虽然数据结构有很多，比如树，图，哈希表等。但真正的实现还需要落实到具体的基础数据结构，即**数组和链表**。之所以说他们是基础的数据结构，是因为它们直接控制物理内存的使用。

数组使用连续的内存空间，来存储一系列同一数据类型的值。如图表示的是数组的**每一项都使用一个 byte 存储**的情况。



那么为什么数组要存储相同类型的值呢？为什么有的语言（比如 JS）就可以存储不同类型的值呢？

实际上存储相同的类型有两个原因：

1. 相同的类型**大小是固定且连续的**(这里指的是基本类型，而不是引用类型，当然引用类型也可以存一个大小固定的指针，而将真实的内容放到别的地方，比如内存堆)，这样数组就可以**随机访问**了。试想数组第一项是 4 字节，第二项是 8 字节，第三项是 6 字节，我如何才能随机访问？而如果数组元素的大小都一样，我们就可以用**基址 + 偏移量**来定位任意一个元

素，其中基址指的是数组的引用地址，如上图就是 1001。 偏移量指的是数组的索引 * 数组每一项所占用的内存空间大小。

2. 静态语言要求指定数组的类型。

> 虽然在一些语言，比如 JavaScript 中，数组可以保存不同类型的值，这是因为其内部做了处理。对于 V8 引擎来说， 它将数据类型分为基本类型和引用类型，基本类型直接存储值在栈上，而引用类型存储指针在栈上，真正的内容存到堆上。因此不同的数据类型也可以保持同样的长度。

数组的一个特点就是**支持随机访问**，请务必记住这一点。当你需要支持随机访问的数据结构的话， 自然而然应该想到数组。

本质上，数组是一段连续的地址空间，这个是和我们之后要讲的链表的本质差别。 虽然二者从逻辑上来看都是线性的数据结构。

这里我总结了数组的几个特性，供大家参考：

- 一个数组表示的是一系列的元素
- 数组（static array）的长度是固定的，一旦创建就不能改变（但是可以有 dynamic array）
- 所有的元素需要是同一类型（个别的语言除外）
- 可以通过下标索引获取到所储存的元素（随机访问）。 比如 array[index]
- 下标可以是是 0 到 array.length - 1 的任意整数

当数组里的元素也是一个数组的时候，就可以形成多维数组。例子：

1. 用一个多维数组表示坐标
2. 用一个多维数组来记录照片上每一个 pixel 的数值

> 力扣中有很多二维数组的题目，我一般称其为 `board` 或者 `matrix` ，这样通过名字一眼就能看出其是一个二维数组。

比如后面要讲的动态规划，如果题目的状态不止一个，我们就需要使用多维数组来存储。每一个状态对应数组中的一个维度。另外，后面的图部分，我们很多时候都会用二维数组来建立邻接矩阵。

数组的常见操作

了解了数组的底层之后，我们来看下数组的基本操作以及对应的时间复杂度。

1. 随机访问，时间复杂度 O(1)

```
arr = [1,2,33]
arr[0] # 1
arr[2] # 33
```

2. 遍历，时间复杂度 O(N)

```
for num in nums:
  print(num)
```

3. 任意位置插入元素、删除元素

```
arr = [1,2,33]
# 在索引2前插入一个5
arr.insert(2, 5)
print(arr) # [1,2,5,33]
```

我们不难发现， 插入 2 之后，新插入的元素之后的元素（最后一个元素）的索引发生了变化，从 2 变成了 3，而其前面的元素没有影响。从平均上来看，数组插入元素和删除元素的时间复杂度为 `O(N)` 。最好的情况删除和插入发生在尾部，时间复杂度为 `O(1)` 。

基本上数组都支持这些方法。 虽然命名各有不同，但是都是上面四种操作的实现：

- each()： 遍历数组
- pop(index)：删除数组中索引为 index 的元素
- insert(item, index)：数组索引为 index 处插入元素

时间复杂度分析小结

- 随机访问 -> O(1)
- 根据索引修改 -> O(1)
- 遍历数组 -> O(N)
- 插入数值到数组 -> O(N)
- 插入数值到数组最后 -> O(1)
- 从数组删除数值 -> O(N)
- 从数组最后删除数值 -> O(1)

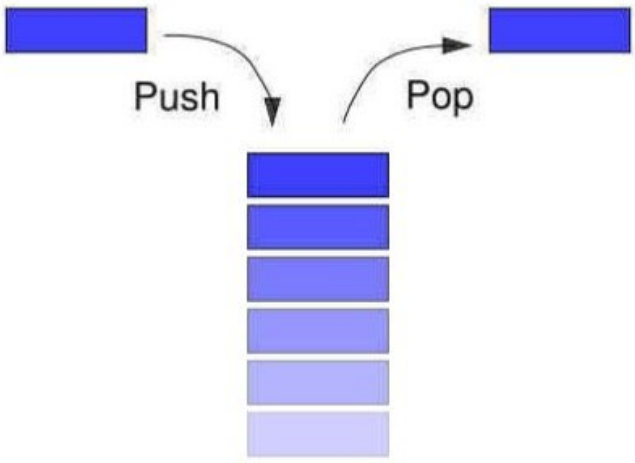
题目推荐

- **414. 第三大的数** 强烈推荐 👍
- **剑指 Offer 53 - II. 0 ～ n-1 中缺失的数字** 强烈推荐 👍
- **88. 合并两个有序数组** 强烈推荐 👍
- **380. 常数时间插入、删除和获取随机元素** 强烈推荐 👍
- **41. 缺失的第一个正数**

另外推荐两个思考难度小，但是边界多的题目， 这种题目如果可以一次写出 bug free 的代码会很加分。

- **59. 螺旋矩阵 II**
- **859. 亲密字符串**

栈



栈是一种受限的数据结构， 体现在**只允许新的内容从一个方向插入或删除，这个方向我们叫栈顶**，另一端一般称为栈底。除了栈顶的其他位置获取或操作内容都是不被允许的。

栈最显著的特征就是 LIFO(Last In, First Out - 后进先出)

举个例子：

栈就像是一个放书本的抽屉，进栈的操作就好比是想抽屉里放一本书，新进去的书永远在最上层，而出栈则相当于从里往外拿书本，永远是从最上层开始拿，所以拿出来的永远是最后进去的哪一个。

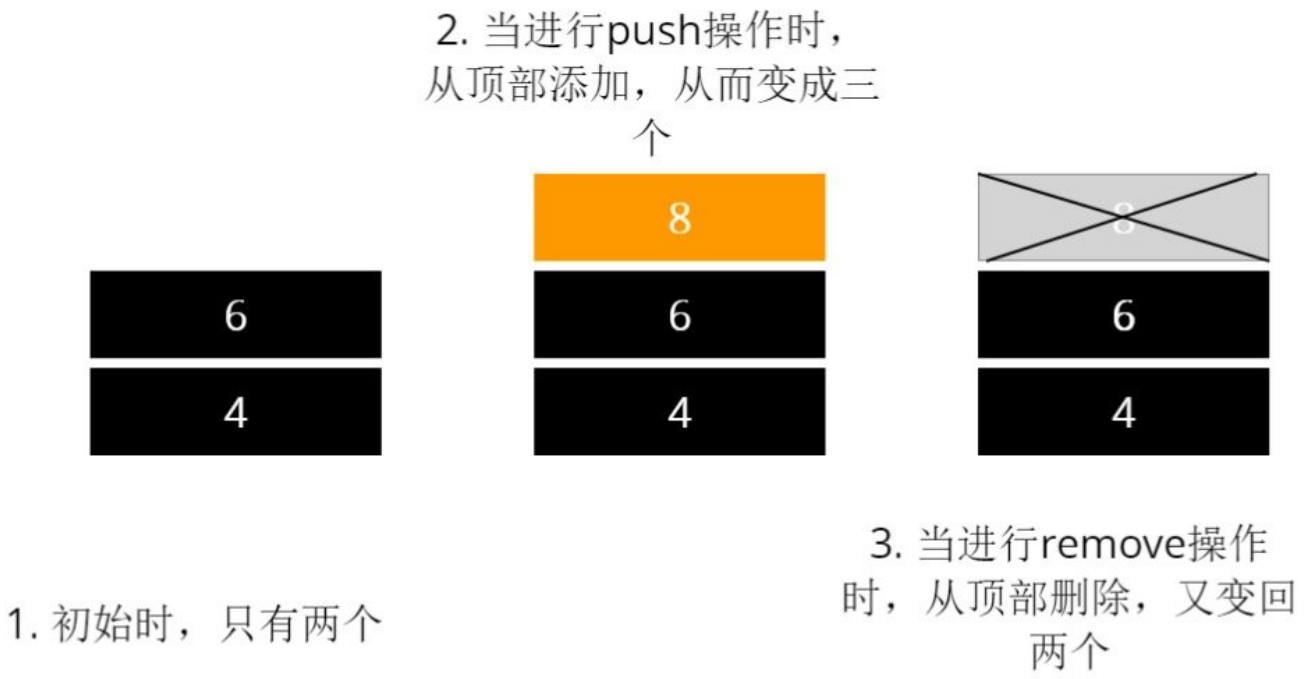
栈的常用操作与时间复杂度

- 进栈 - push - 将元素放置到栈顶

- 出栈 - pop - 将栈顶元素弹出
- 取栈顶 - top - 得到栈顶元素的值
- 判断是否为空栈 - isEmpty - 判断栈内是否有元素

复杂度分析：

- 进栈 - O(1)
- 出栈 - O(1)
- 取栈顶 - O(1)
- 判断是否为空栈 - O(1)



实现

由于栈只允许在尾部操作，我们用数组进行模拟的话，可以很容易达到 O(1)的时间复杂度。

> 当然也可以用链表实现，即链式栈。

我们可以使用一个数组加一个变量（记录栈顶的位置）的方式很方便的实现栈。实现过程也非常简单，即将数组的 API 删除几个就好了。

比如数组支持在头部添加和删除元素以及遍历数组等 API，我们将其删除就可以直接将其看成是栈。这也充分应证了我开头的话**栈是一种受限的数据结构**。

应用

栈是实现深度优先遍历的基础。除此之外，栈的应用还有很多，这里列举几个常见的。

- 函数调用栈
- 浏览器前进后退
- 匹配括号
- 单调栈用来寻找下一个更大（更小）元素 推荐题目：**Every-Sublist-Min-Sum**

除此之外，有两个在数学和计算机都应用超级广泛的就是是 波兰表示法 和 逆波兰表示法 ，之所以叫波兰表示法，是因为其是波兰人发明的。

波兰表示法（Polish notation，或波兰记法），是一种逻辑、算术和代数表示方法，其特点是操作符置于操作数的前面，因此也称做前缀表示法。如果操作符的元数（arity）是固定的，则语法上不需要括号仍然能被无歧义地解析。波兰记法是波兰数学家扬·武卡谢维奇 1920 年代引入的，用于简化命题逻辑。

扬·武卡谢维奇本人提到：[1]

> “ 我在 1924 年突然有了一个无需括号的表达方法，我在文章第一次使用了这种表示法。 ”

以下是不同表示法的直观差异：

- 前缀表示法 （+ 3 4 ）
- 中缀表示法 （3 + 4）
- 后缀表示法 （3 4 + ）

LISP 的 S-表达式中广泛地使用了前缀记法，S-表达式中使用了括号是因为它的算术操作符有可变的元数（arity）。逆波兰表示法在许多基于堆栈的程序语言（如 PostScript）中使用，以及是一些计算器（特别是惠普）的运算原理。

题目推荐

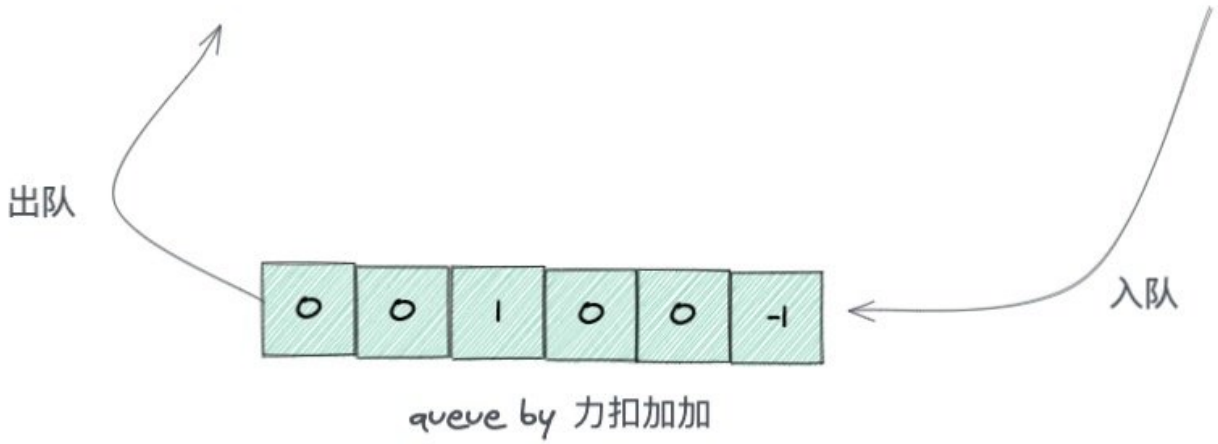
- **946. 验证栈序列** 🔥🔥🔥 强烈推荐
- **150. 逆波兰表达式求值**
- **1381. 设计一个支持增量操作的栈**
- **394. 字符串解码**

另外还有**两个计算器的题目**也值得练习。

队列

同样地，队列也是一种受限的数据结构。

和栈相反，队列是只允许在**一端**进行插入，在**另一端**进行删除的线性表。因此队列(Queue)是一种先进先出(FIFO - First In First Out)的数据结构，通常情况下，我们称队列中插入元素的一端为尾部，删除元素的一端为头部。



队列也是一种逻辑结构，底层同样可以用数组实现，也可以用链表实现，不同实现有不同的取舍。

如果用数组实现，那么入队或者出队的时间复杂度一定有且仅有一个是$O(N)$的，其中 N 为队列的长度。而使用链表实现则可以在 $O(1)$ 的时间完成任何合法的队列操作。这得益于链表对动态添加和删除的友好性。关于链表的队列的实现，我们会在后面的 `队列的实现（Linked List）` 部分讲解。

队列的操作与时间复杂度

- 插入 - 在队列的尾部添加元素
- 删除 - 在队列的头部删除元素
- 查看首个元素 - 返回队列头部的元素的值

时间复杂度取决于你的底层实现是数组还是链表。我们知道直接用数组模拟队列的话， 在队头删除元素是无法达到 O(1) 的复杂度的， 上面提到了由于存在调整数组的原因，时间复杂度为 `O(N)` 。因此我们需要一种别的方式，这种方式就是下面要讲的 Linked List。

以链表为例，其时间复杂度：

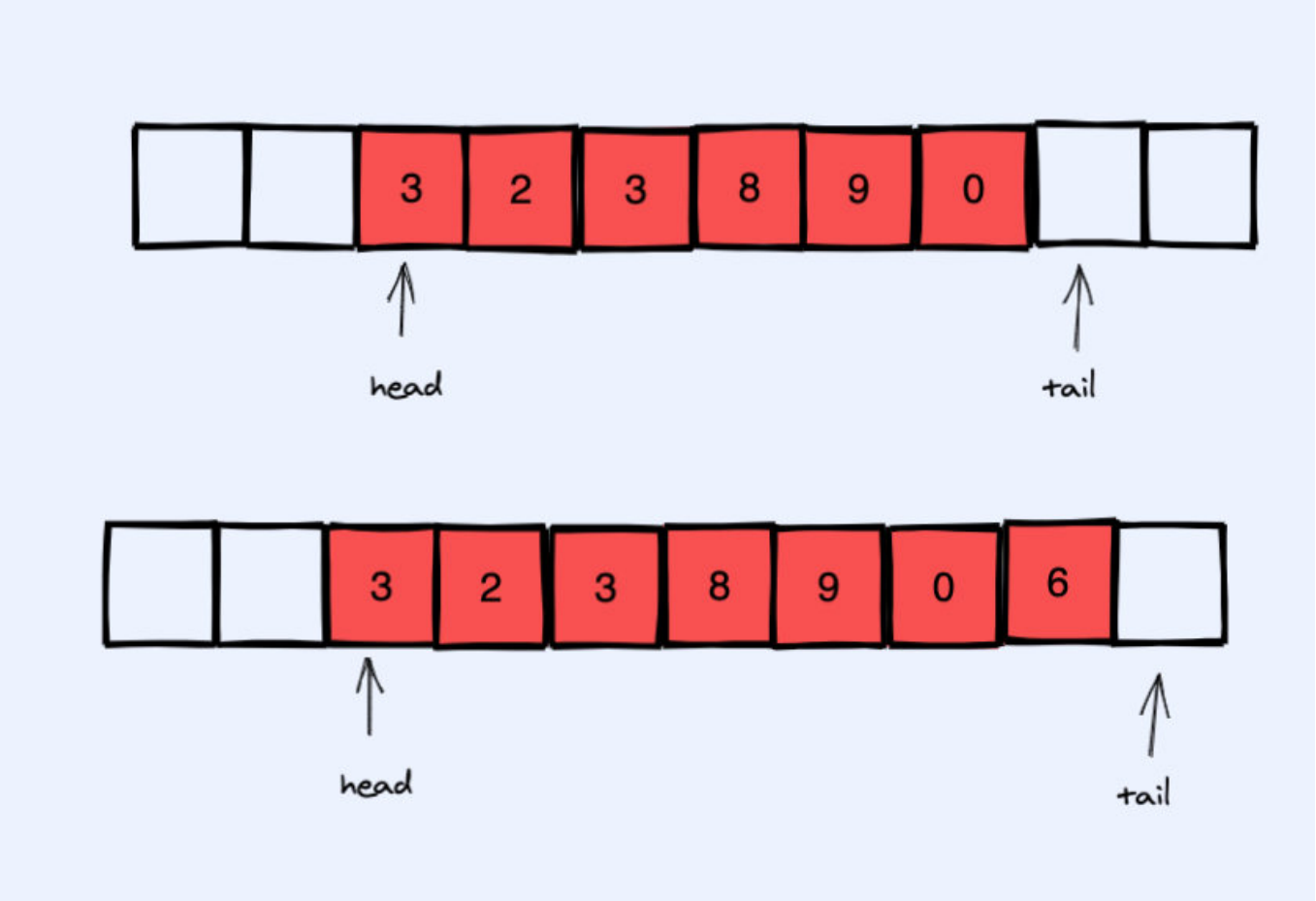
- 插入 - O(1)
- 删除 - O(1)

- 查看首个元素 - O(1)

实际上队列也可用数组来实现，并且插入和删除时间复杂度都是 (1)，如何实现呢?

其实我们值需要用两个指针 head 和 tail 表示队列的头和尾部，然后给队列分别一个空间。当插入的时候，我们在 tail 的前一个内存单元插入一个值，并更新 tail 即可。

如图是在一个队列中插入一个数字 6 的内部情况。



在头部删除也是类似的。

这种实现的方式，需要动态开辟内存。如果不考虑内存不够而 copy 内存的情况，那么整体的时间复杂度可以控制在 $O(1)$。

> CPP 的 deque 就是这么实现的

应用

队列的应用同样广泛。在做题中最主要的一个应用就是**广度优先遍历**（BFS）。在工程中同样使用广泛，比如消费队列，浏览器的 HTTP 请求队列等等。

队列的实现（Linked List）

我们知道链表的删除操作，尤其是删除头节点的情况下，是很容易做到 O(1) 的时间复杂度的。

那么我们是否可利用这一点来弥补上面说的删除无法达到 O(1) 时间复杂度呢?

> 删除非头节点可以做到 O(1)么？什么情况下可以?

但是在链表末尾插入需要遍历到尾部的话就不是 O(1)，而是 O(N)了。

解决这个问题其实不复杂，只要维护一个变量 tail， 存放当前链表的尾节点引用即可在 `O(1)` 的时间完成插入操作。

因此使用链表进行模拟的话。

入队就是：

```
tail.next = newNode()
tail = newNode()
```

类似地，我们维护一个 head 虚拟节点也可是在 $O(1)$ 时间出队。

出队就是：

```
nnext = head.next.next

head.next = nnext
```

具体的代码大家可以在学习完链表章节在回头补充。

另外大家在平时做题的时候可以直接使用内置的队列，比如 Python 的 deque。

> 严格意义上 deque 是双端队列，其允许在两端同时进行插入和删除。因此比普通队列的操作更宽松，不是严格的队列。不过和栈类似，我们删除几个 API 就可以将其看成是一个基于链表实现的队列了。

除此之外，还有一种队列是循环队列，用的不是很多。篇幅所限，不在这里展开，感兴趣的可以自己查一下。

推荐题目

- Quadratic-Application 不知道啥时候应该用队列？这题可以看一下。

技巧

就位

有些题目限制了数组的数据范围是 1 - n（其实值域量和数组长度一致就 ok），并且让你找符合条件的值就可能会使用到这个技巧 。

比如如下题目：

```
给你一个长度为 n 的整数数组 nums ，其中 nums 的所有整数都在范围 [1, n] 内，且每个整数出现 一次 或 两次 。请你找出所有出现 两次 的整数，并以
```

我们可以将数组中的每一个元素映射到正确位置。 比如让 1 就位到 nums[0]，2 就位到 nums[1]，3 就位到 nums[2]，以此类推。 这样就可以得到一个新的数组，其中每个元素都是正确位置的元素。最后我们只需要再次遍历这个数组，找到**没有就位**的元素，在这里就是 i + 1 != nums[i] ，就可以找出所有出现两次的元素。

我们可以把就位的过程看成是寻找真爱的过程。 在这里 nums[i] 的真爱就是 i + 1。

我们遍历一次数组 nums， 如果 nums[i] != i + 1 ，说明 nums[i] 没有找到真爱。我们不知道 i + 1 位于数组哪一项，没有办法交换。但是**我们知道 nums[i]就是 nums[i] - 1 的真爱， 所以我们可以和其交换，成全他人，这样 nums[i] - 1 就找到真爱了**。重复这个过程，直到 nums[i] != nums[nums[i] - 1]， 此时继续交换将会无限循环。

代码参考：

```python
class Solution:
    def findDuplicates(self, nums: List[int]) -> List[int]:
        ans = []
        # 就位
        for i in range(len(nums)):
            while nums[i] !=  nums[nums[i] - 1]:
                nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]
        # 再次遍历到目标
        for i in range(len(nums)):
            if nums[i] != i + 1: ans.append(nums[i])
        return ans
```

推荐题目：

- **41. 缺失的第一个正数** 强烈推荐 👍
- **442. 数组中重复的数据**

相关专题

前缀和

关于前缀和， 看我的这篇文章就够了 ~ **【西法带你学算法】一次搞定前缀和**

单调栈

单调栈适合的题目是求解**第一个大于 xxx**或者**第一个小于 xxx**这种题目。所以当你有这种需求的时候，就应该想到**单调栈**。

栈是**最难**的数据结构，而栈中单调栈的使用又是**最难**之一，因此值得大家投入更多的精力。

单调栈的使用实在是太多了，我之前写的一个系列题解核心也就是单调栈。参考： **一招吃遍力扣四道题，妈妈再也不用担心我被套路啦～**。

单调队列和单调栈的思路比较类似，感兴趣的可以自己查阅一下相关资料作为扩展。这里只推荐一道题 **239. 滑动窗口最大值**。

下面两个题帮助你理解单调栈， 并让你明白什么时候可以用单调栈进行算法优化。

- **84. 柱状图中最大的矩形**
- **739.每日温度**

另外再推荐两道题，这两道题都是局部有序性，有有序性往往和单调栈有联系。

- **255. 验证前序遍历序列二叉搜索树**
- **768. 最多能完成排序的块 II**

单调队列也是类似的，只不过数据结构变成了队列而已。这里推荐一道题。

- **Longest Equivalent Sublist After K Increments**

这道题需要维护窗口内的最大值，因此可以考虑单调队列来维护。相似的题目力扣也有 **239. 滑动窗口最大值**，大家可以对比练习。

栈匹配

当你需要比较类似栈结构的匹配的时候，就应该想到使用栈。

比如判断有效括号。 我们知道有效的括号是形如： `(((()))` 这样的括号， 其中第一个左括号和最后一个右括号匹配，因此一种简单的思路是把左括号看出是入栈，右括号看出是出栈即可轻松利用栈的特性求解。·

再比如链表的回文判断。 我们就可以一次遍历压栈，再一次遍历出栈的同时和当前元素比较即可。这也是利用了栈的特性。

推荐几个经典的题目：

- **20. 有效的括号**
- **678. 有效的括号字符串**
- **2116. 判断一个括号字符串是否有效** 如果把没有锁定的括号看做是 `*` ，那么和上面的 678 是一样的。

计数

从代码上看，我们通常会建立一个 counts 数组来计数，其本质和 Python 的 collections.Counter 类似。

比如对字符串 "abac" 中的字母计数，那么结果就是 a 有 2 个， b 和 c 分别有 1 个。

对于上面的问题，我们可以开辟一个长度为 26 的数组 counts，其中 counts[0] 用来表示 a 的出现次数，其中 counts[1] 用来表示 b 的出现次数，以此类推。

实现起来比较简单，代码：

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        str_dict = collections.defaultdict(list)
        for s in strs:
          s_key = [0] * 26
          for c in s:
            s_key[ord(c)-ord('a')] += 1
          str_dict[tuple(s_key)].append(s)
        return list(str_dict.values())
```

49.字母的异位词分组，825. 适龄的朋友 以及 【每日一题】 Largest Range 等就是计数，分桶思想的应用。力扣关于分桶思想的题目有很多，大家只要多留心就不难发现。

总结

数组和链表是最最基础的数据结构，大家一定要掌握，其他数据结构都是基于两者产生的。

栈和队列是两种受限的数据结构，我们人为地给数组和链表增加一个限制就产生了它们。那我们为什么要自己给自己设限制呢？目的就是为了简化一些常见问题，这就好像是人类模仿鸟制造了飞机，模仿鸽子做了地震仪一样。栈和队列能帮我们简化问题。 比如队列的特性就很适合做 BFS，栈的特性就很适合做括号匹配等等。你可以这么理解。 我们一开始做 BFS 的时候，没有队列。慢慢大家写地多了，发现是不是可以**抽象一个数据结构单独来处理这种通用的需求**？队列就产生了，其他数据结构也是一样。

参考

- 基础数据结构 by lucifer