介绍

栈、队列和Hash在工程里有大量的应用，而在算法中是相对简单的内容，而且在很多时候，这三个结构都是解决更高级问题时的工具，可以直接用的。因此我们首先要理解这几个技术的特征、实现原理以及如何在算法中应用等问题。

栈的典型面试题有：表达式求解、括号分析相关的问题、计算器相关的问题、逆波兰表达式等。这些问题略有复杂，重要性与后续要学习的树等相比不算高，所以如果时间不够，可以暂时跳过， 后面再学习。

至于队列，直接考队列的算法题几乎没有，大部分场景是作为高级算法的一个工具，典型的问题是树里的层次遍历相关问题和图等高级主题中与广度优先相关的问题。而队列真正的大热门是作为技术面试，考察JUC里的阻塞队列、AQS等的实现原理等等， 这个一般在多线程相关课程里会深入讲解，我们以面试的需要为主，不再赘述。

Hash，不管是算法，还是在工程中都会大量使用。很多复杂的算法问题用Hash能轻松解决，也正是如此，在算法里就显得没什么思维含量，所以Hash是应用里的扛把子，但在算法里就是备胎的角色，只要有其他方式一般就不考虑了。这也是面试算法与应用算法的一个区别。

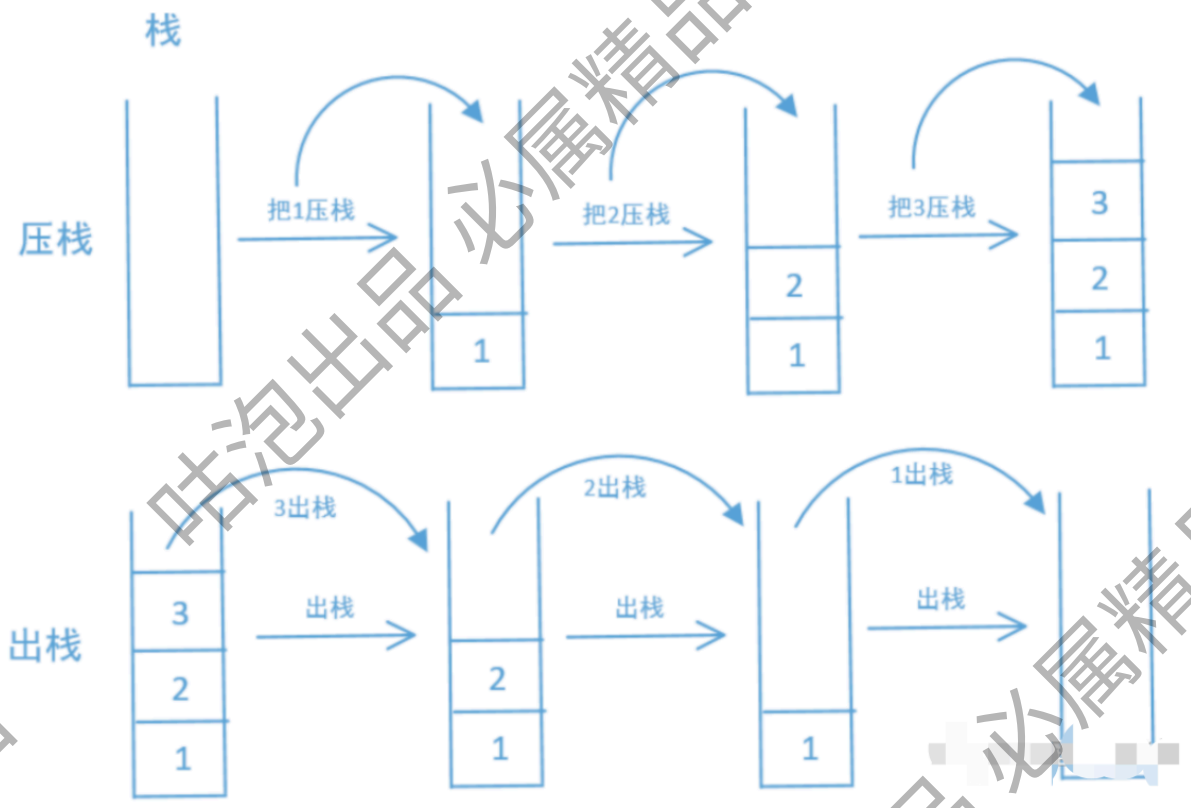
另外基于本章的内容，可以产生大量的设计型的算法，例如让你设计跳表、计数器、计算器、缓存设计等等，这类的题目一般难度不大，重点在于设计合理、编码规范、考虑周全。有些面试官不喜欢烧脑的算法，就喜欢考察这类问题。设计类型的题目我们后面有单独的一章来介绍。

1.栈基础知识

1.1 栈的特征

栈和队列是比较特殊的线性表，又称之为访问受限的线性表。栈是很多表达式、符号等运算的基础，也是递归的底层实现。理论上递归能做的题目栈都可以，只是有些问题用栈会非常复杂。

栈底层实现仍然是链表或者顺序表，栈与线性表的最大区别是数据的存取的操作被限制了，其插入和删除操作只允许在线性表的一端进行。一般而言，把允许操作的一端称为栈顶(Top)，不可操作的一端称为栈底(Bottom)，同时把插入元素的操作称为入栈(Push),删除元素的操作称为出栈(Pop)。若栈中没有任何元素，则称为空栈，栈的结构如下图：



1.2 栈的操作

栈的常用操作主要有：

```
Stack() 创建一个新的空栈
push(item) 添加一个新的元素item到栈顶
pop() 弹出栈顶元素
peek() 返回栈顶元素
is_empty() 判断栈是否为空
size() 返回栈的元素个数
```

我们在设计自己的栈的时候，不管用数组还是链表，都要实现上面几个方法。如果想测试一下自己对栈是否理解，做一下这道题就够了：**入栈顺序为1234,所有可能的出栈序列是什么?**

4个元素的全排列共有24种，栈要求符合后进先出，按此衡量排除后即得：

1234√ 1243√ 1324√ 1342√ 1423× 1432√

2134√ 2143√ 2314√ 2341√ 2413× 2431√

3124× 3142× 3214√ 3241√ 3412× 3421√

4123× 4132× 4213× 4231× 4312× 4321√

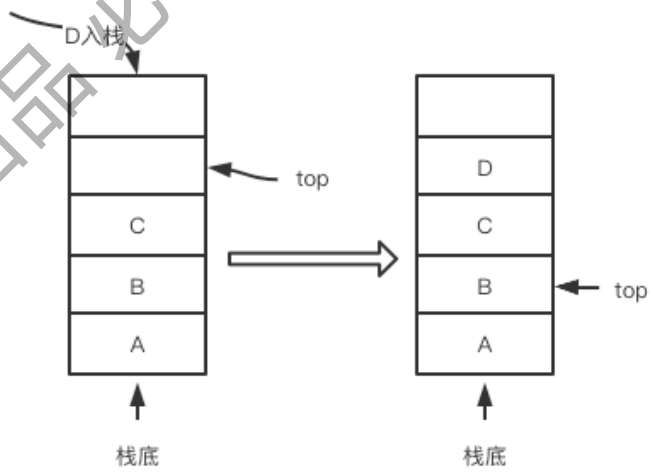
14种可能，10种不可能，如上所示。

1.3 栈的基本实现

如果要自己实现栈，可以有数组和链表两种基本的方式。在python中有一个比较好用的数据结构-list，我们基于此来实现栈。

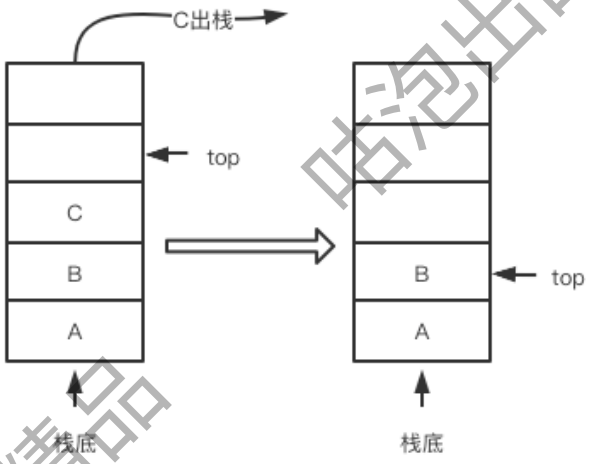
采用顺序表实现的的栈，内部以数组为基础，实现对元素的存取操作。在应用中还要注意每次入栈之前先判断栈的容量是否够用，如果不够用，可以用Arrays.copyOf()进行扩容。

入栈过程如下所示：



top指向的是栈顶元素，新元素入栈时先top++，然后再将元素入栈。

出栈的过程如图所示：



top先将栈顶元素取出，然后再执行top--。完整的实现代码是：

```
# coding=utf-8
```

```
class Stack(object):
    """栈"""

    def __init__(self):
        self.items = []

    def is_empty(self):
        """判断是否为空"""
        return self.items == []

    def push(self, item):
        """加入元素"""
        self.items.append(item)

    def pop(self):
        """弹出元素"""
        return self.items.pop()

    def peek(self):
        """返回栈顶元素"""
        return self.items[len(self.items) - 1]

    def size(self):
        """返回栈的大小"""
        return len(self.items)

if __name__ == "__main__":
    stack = Stack()
    stack.push("hello")
    stack.push("world")
    stack.push("itcast")
    print stack.size()
    print stack.peek()
    print stack.pop()
    print stack.pop()
    print stack.pop()
```

栈的典型题目还是非常明显的，括号匹配、表达式计算等等几乎都少不了栈，而且难度还比较大，详细看本文第5节。

2. 队列基础知识

2.1.队列的概念和实现

队列的特点是节点的排队次序和出队次序按入队时间先后确定，即先入队者先出队，后入队者后出队，即我们常说的FIFO(first in first out)先进先出。队列实现方式也有两种形式，基于数组和基于链表。

队列的主要操作有如下几个：

- Queue() 创建一个空的队列
- enqueue(item) 往队列中添加一个item元素
- dequeue() 从队列头部删除一个元素
- is_empty() 判断一个队列是否为空
- size() 返回队列的大小

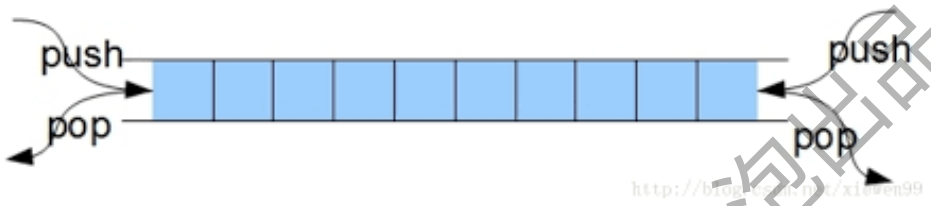
在python中实现队列还是比较简单的;

```
class Queue(object):
    """队列"""

    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def enqueue(self, item):
        """进队列"""
        self.items.insert(0, item)

    def dequeue(self):
        """出队列"""
        return self.items.pop()

    def size(self):
        """返回大小"""
        return len(self.items)

if __name__ == "__main__":
    q = Queue()
    q.enqueue("hello")
    q.enqueue("world")
    q.enqueue("itcast")
    print q.size()
    print q.dequeue()
    print q.dequeue()
    print q.dequeue()
```

2.2 双端队列

双端队列（deque，全名double-ended queue），是一种具有队列和栈的性质的数据结构。

双端队列中的元素可以从两端弹出，其限定插入和删除操作在表的两端进行。双端队列可以在队列任意一端入队和出队。



双端队列在实际中有非常广泛的应用，Java和python都有为其提供了强大的功能。其常见操作是;

- Deque() 创建一个空的双端队列
- add_front(item) 从队头加入一个item元素
- add_rear(item) 从队尾加入一个item元素
- remove_front() 从队头删除一个item元素
- remove_rear() 从队尾删除一个item元素
- is_empty() 判断双端队列是否为空
- size() 返回队列的大小

实现：

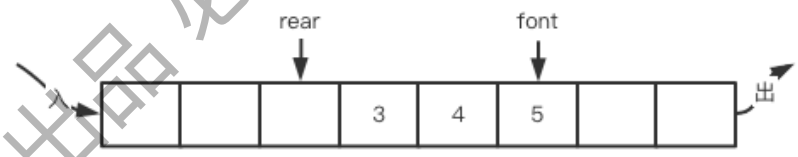
```
# coding=utf-8
class Deque(object):
    """双端队列"""

    def __init__(self):
        self.items = []

    def is_empty(self):
        """判断队列是否为空"""
        return self.items == []

    def add_front(self, item):
        """在队头添加元素"""
        self.items.insert(0, item)

    def add_rear(self, item):
        """在队尾添加元素"""
        self.items.append(item)

    def remove_front(self):
        """从队头删除元素"""
        return self.items.pop(0)

    def remove_rear(self):
        """从队尾删除元素"""
        return self.items.pop()

    def size(self):
        """返回队列大小"""
        return len(self.items)

if __name__ == "__main__":
```

```
deque = Deque()
deque.add_front(1)
deque.add_front(2)
deque.add_rear(3)
deque.add_rear(4)
print deque.size()
print deque.remove_front()
print deque.remove_front()
print deque.remove_rear()
print deque.remove_rear()
```

2.3 环形队列

在上面的代码中，items.insert()和items.pop()方法隐藏了一个很重要的细节，那就是循环访问的问题，该问题在C/C++和java等语言中都要注意的，我们来看一下。

如果是基于数组的，会有点麻烦，例如 假如我们初始化一个长度是7的队列：



顺序存储结构存储的队列称为顺序队列，内部使用一个一维数组存储，用一个队头指针front指向队列头部节点（即使用int类型front来表示队头元素的下标），用一个队尾指针rear(有的地方会用tail，只要在一个问题里统一起来就行了)，指向队列尾部元素(int类型rear来表示队尾节点的下标)。

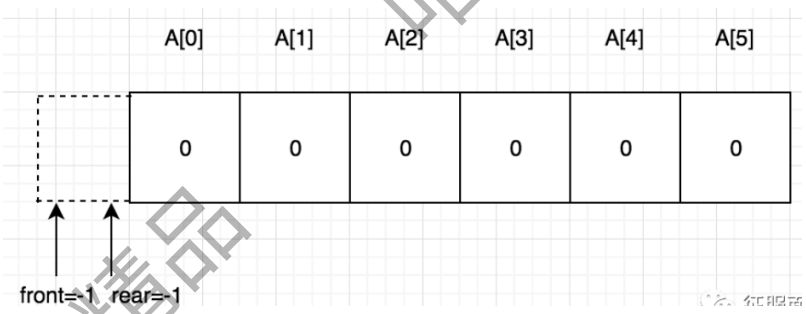
初始化队列时: front = rear = -1 (非必须，也可设置初始值为0，在实现方法时具体修改)

队列满时：rear = maxSize-1 (其中maxSize为初始化队列时，设置的队列最大元素个数)

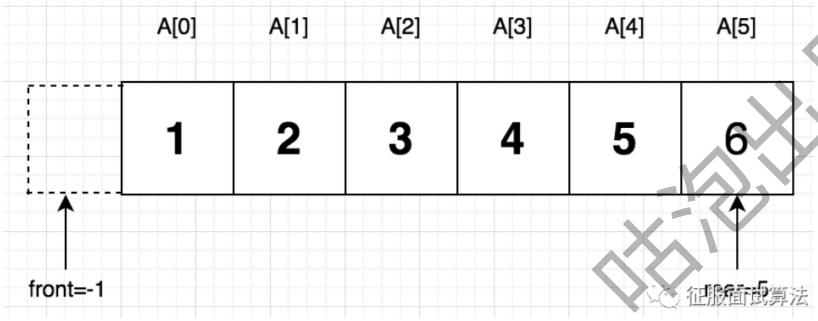
队列为空时：front = rear

front指向的是队列的头，rear指向的是队列尾的下一个存储空间，最初始的时候front=0， rear=0，每添加一个元素rear就加1，每移除一个元素front就加1，但是这样会有一个问题，如果一个元素不停的加入队列，然后再不停的从队列中移除，会导致rear和front越来越大，最后会导致队列无法再加入数据了，但实际上队列前面全部都是空的，这导致空间的极大浪费。

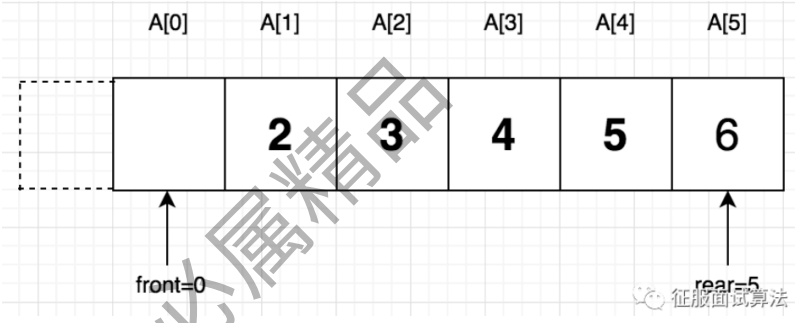
在代码中初始化了一个大小为6的顺序队列，下图展示了第一步(即代码ArrayQueue queue = new ArrayQueue(6))中队列元素及指针情况：



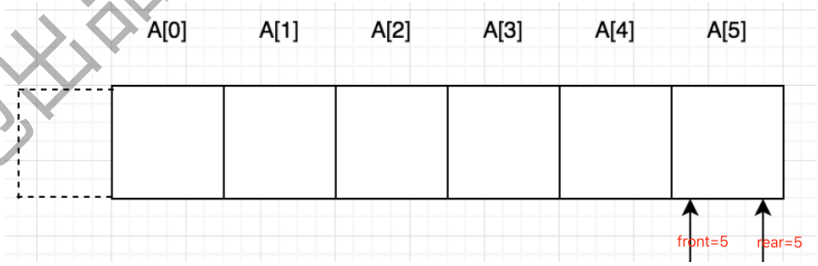
其中front和rear指向的虚线框实际并不存在，仅用来表示初始化时的默认状态，因为我们实现的队列元素使用int[]存储元素，所以初始值均为0(如用Object[]或范型则初始值为null)，执行queue.add(1)到queue.add(6)方法后队列的状态如下图：



接下来看下队列的出队情况，当第一次执行queue.pop()方法后，队列元素如上图所示，此时队列剩下5个元素：



当第六次执行queue.pop()方法后，队列元素如下图所示：



此时队列中元素已全部出队，按正常逻辑应该可以添加元素到队列中，但此时添加元素却会报队列已满错误（rear=maxSize-1），当然即使前面元素未出队也会报相同错误。这就是我们常说的“假溢出”问题。为解决这个问题，就引出了我们的环形队列。

在我们上面基于数组实现的队列中，假如头和尾都到了末尾，接下来还有新元素来该怎么办呢？虽然这时候个空间是空的，但是无法插入新元素，为此，我们将其设计成环形结构。

环形队列，顾名思义即让普通队列首尾相连，形成一个环形。当rear指向尾元素后，当队列有元素出队时，可以继续向队列中添加元素。这里使用rear指针指向最后一个节点的后一个元素，即会占用一个位置用来表示队列已满。

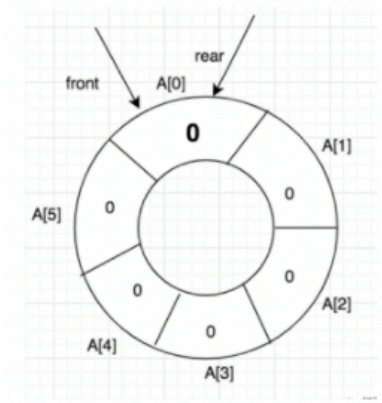
- 初始化队列时: front = rear = 0
- 队列满时： （rear +1) % maxSize == front (其中maxSize为初始化队列时，设置的队列最大元素个数)

这里不能使用rear = maxSize-1作为判断队满的条件，因使用环形队列方式实现，当第一次队满时，rear = maxSize -1，执行出队操作后原队头位置空出，此时继续执行入队操作，则rear向后移动一个位置，则rear = 0，而此时队列也是已满状态。所以只要rear 向前移动一个位置就等于front时，就是队满的情况。

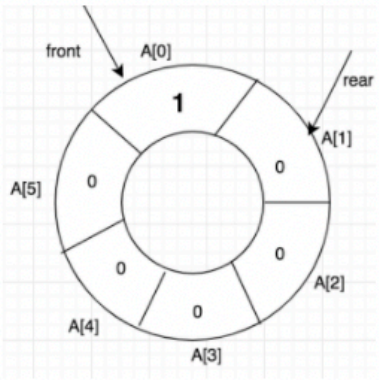
* 队列为空时：front == rear。

下面再以图解的方式讲解一下环形队列的入队出队以及队满情况。图A是初始状态，此时front = rear = 0，队列为空。当第一次执行queue.add(1)后，环形队列元素如图B所示。

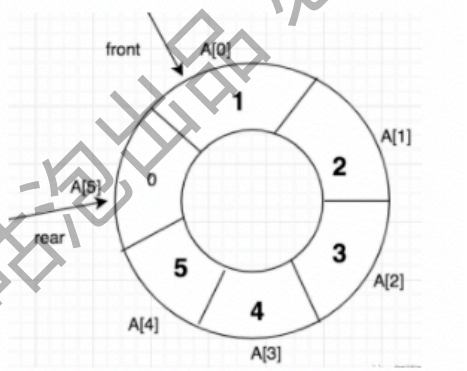
当依次执行queue.add(2);queue.add(3);queue.add(4);queue.add(5);后，达到(rear+1)%maxSize=front （即rear=5）条件，队列已满不能添加新元素。此时环形队列元素情况如图C所示：



图A 初始状态



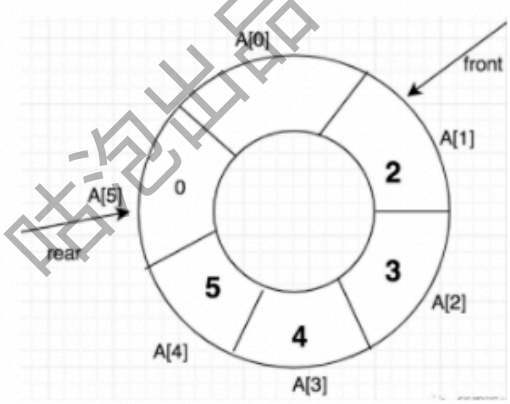
图B 增加一个元素



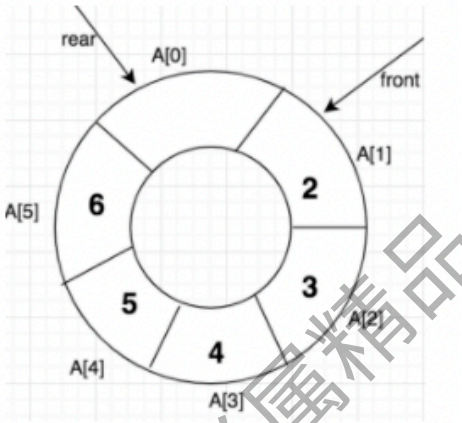
图C 连续增加元素，队满

所以这种方式会浪费一个空间来作为判满的条件。

下面执行出队操作，当第一次执行出队操作queue.pop()方法后，环形队列元素情况如下图A所示。此时(rear+1)%maxSize = 0 不等于 front=1 ,所以可以继续向队列中添加元素，也就不会出现假溢出的情况。当执行入队(例queue.add(6))操作后，rear = (rear+1)%maxSize 即rear=0，以此来生成环形队列。此时队列元素情况如下图B所示：



图A 元素1出队



图B 元素6入队

在这种场景下，环形队列有效元素个数该怎么计算呢，如果不是环形队列，则有效元素个数size = rear - front。而使用环形实现后，会出现rear<front的情况，所以要使用(rear-front+maxSize)%maxSize的方式计算有效元素个数。（或者在内部定义一个size属性，当元素入队时size++，当出队时size--）。因此在打印队列中元素时，从front位置开始至 front+size位置结束来循环打印有效元素。

如果不实用环形队列方式实现队列，则会出现“假溢出”情况（即队列满后，将全部元素出队却不能继续添加元素的情况）。而环形队列会在队头元素出队后，将队尾指针rear重新分配为0，以达到循环使用队列空间的目的。

基于数组实现的例子：

```
MaxSize = 100  # 全局变量，假设容量为100
class CircleQueue:  # 循环队列类
    def __init__(self):  # 构造方法
        self.data = [None] * MaxSize  # 存放队列中元素
        self.front = 0  # 队头指针
        self.rear = 0  # 队尾指针
```

```
    def empty(self):  # 判断队列是否为空
        return self.front == self.rear

    def push(self, e):  # 元素e进队
        assert (self.rear + 1) % MaxSize != self.front  # 检测队满
        self.rear = (self.rear + 1) % MaxSize
        self.data[self.rear] = e

    def pop(self):  # 出队元素
        assert not self.empty()  # 检测队空
        self.front = (self.front + 1) % MaxSize
        return self.data[self.front]

    def gethead(self):  # 取队头元素
        assert not self.empty()  # 检测队空
        head = (self.front + 1) % MaxSize  # 求队头元素的位置
        return self.data[head]

    # 例3.11增加的方法
    def size(self):  # 返回队中元素个数
        return (self.rear - self.front + MaxSize) % MaxSize

    def pushk(qu, k, e):
        n = qu.size()
        if k < 1 or k > n + 1:
            return False
        if k <= n:
            for i in range(1, n + 1):
                if i == k:
                    qu.push(e)
                    x = qu.pop()
                    qu.push(x)
                else:
                    qu.ush(e)
        return True

    def popk(qu, k):
        n = qu.ize()
        assert 1 <= k <= n
        for i in range(1, n - 1):
            x = qu.pop()
            if i != k:
                qu.push(x)
            else:
                e = x  # 取第k个出队的元素
        return e

if __name__ == '__main__':
```

```
qu = CircleQueue()
qu.push(1)
qu.push(2)
qu.push(3)
print("元素个数=%d" % (qu.size()))
while not qu.empty():
    print(qu.pop())
print()
print("元素个数=%d" % (qu.size()))
```

3. Hash基础知识

3.1.Hash的概念和基本特征

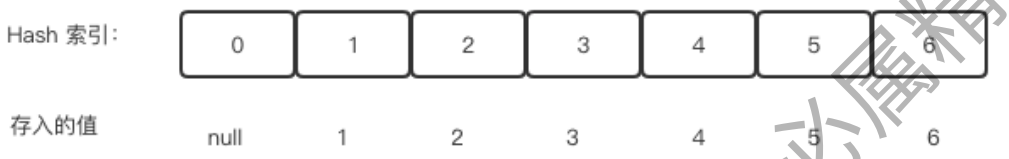
哈希（Hash）也称为散列，就是把任意长度的输入，通过散列算法，变换成固定长度的输出，这个输出值就是散列值。

很多人可能想不明白，这里的映射到底是啥意思，为啥访问的时间复杂度为O(1)？我们只要看存的时候和读的时候分别怎么映射的就知道了。

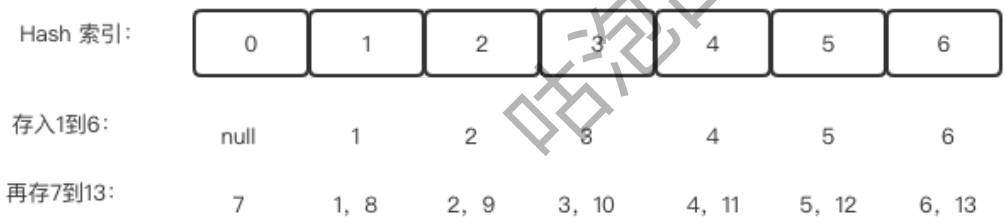
我们现在假设数组array存放的是1到15这些数，现在要存在一个大小是7的Hash表中，该如何存呢？我们存储的位置计算公式是：

$$index = number模7$$

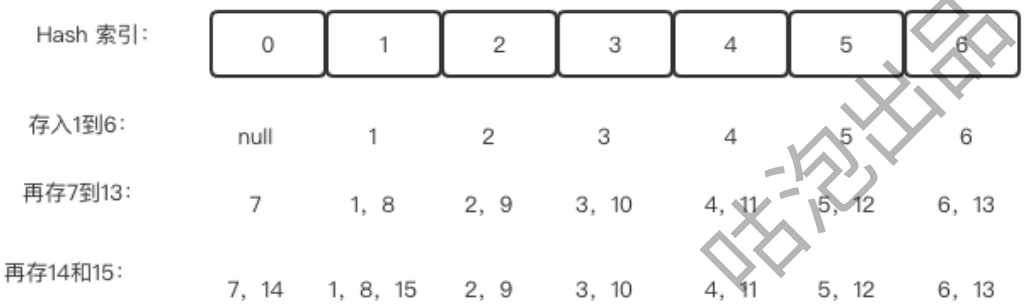
这时候我们将1到6存入的时候，图示如下：



这个没有疑问吧，就是简单的取模。然后继续存7到13，结果是下面这样子：



最后再存14和15:



这时候我们会发现有些数据被存到同一个位置了，我们后面再讨论。接下来，我们看看如何取。

假如我要测试13在不在这里结构里，则同样使用上面的公式来进行，很明显13 模7=6，我们直接访问array[6]这个位置，很明显是在的，所以返回true。

假如我要测试20在不在这里结构里，则同样使用上面的公式来进行，很明显20模7=6，我们直接访问array[6]这个位置，但是只有6和13，所以返回false。

理解这个例子我们就理解了Hash是如何进行最基本的映射的，还有就是为什么访问的时间复杂度为O(1)。

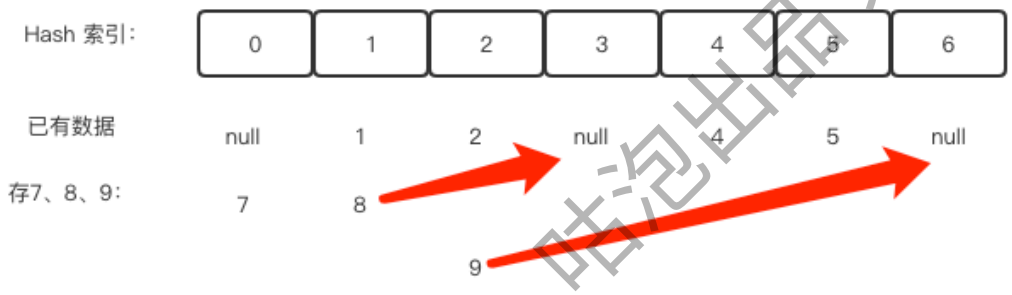
3.2 碰撞处理方法

在上面的例子中，我们发现有些在Hash中很多位置可能要存两个甚至多个元素，很明显单纯的数组是不行的，这种**两个不同的输入值，根据同一散列函数计算出的散列值相同的现象叫做碰撞。**

那该怎么解决呢？常见的方法有：开放定址法(Threadlocal)、链地址法(ConcurrentHashMap)、再哈希法(布隆过滤器)、建立公共溢出区。后两种用的比较少，我们重点看前两个。

3.2.1 开放定址法

开放定址法就是一旦发生了冲突，就去寻找下一个空的散列地址，只要散列表足够大，空的散列地址总能找到，并将记录存入。

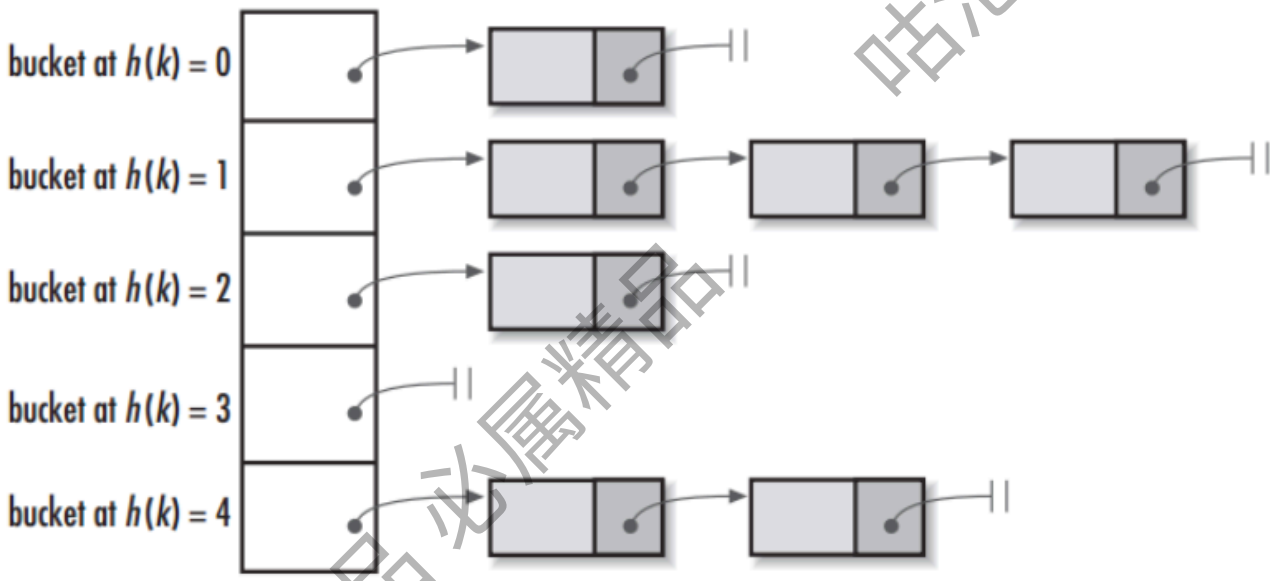


例如上面要继续存7，8，9的时候，7没问题，可以直接存到索引为0位置。8本来应该存到索引为1的位置，但是已经满了，所以继续向后找，索引3的位置是空的，所以8存到3位置。同理9存到索引6位置。

这里你是否有一个疑惑：这样鸠占鹊巢的方法会不会引起混乱？比如再存3 和6的话，本来自己的位置好好的，但是被外来户占领了，该如何处理呢？这个问题解释起来涉及到堆管理的弱引用等概念，简单来说就是在get和set操作的时候如果遇到已经不再使用的元素，则将其清除。而且这种方式一般只适用于元素数量不是特别多的场景。

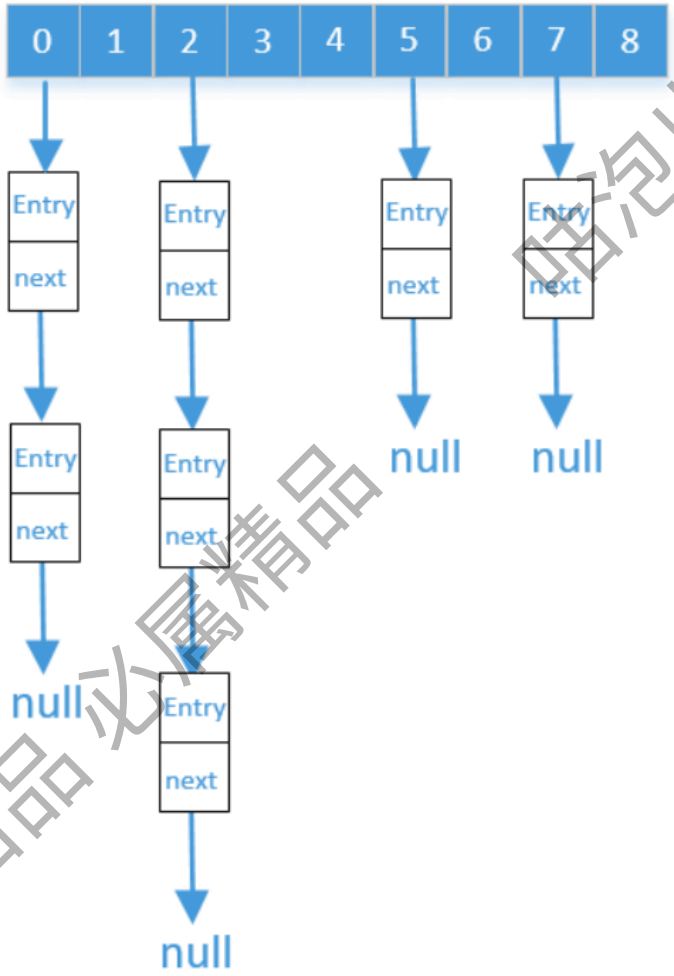
3.2.2 链地址法

将哈希表的每个单元作为链表的头结点，所有哈希地址为i的元素构成一个同义词链表。即发生冲突时就把该关键字链在以该单元为头结点的链表的尾部。例如：

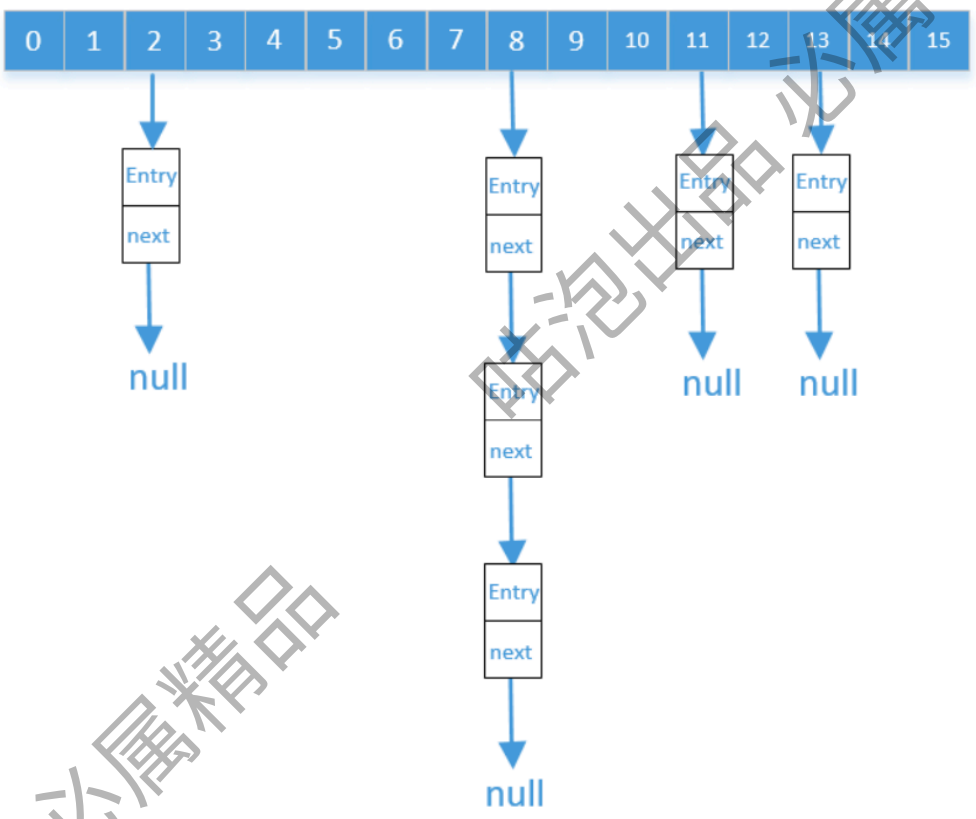


这种处理方法的问题是处理起来代价还是比较高的，要落地还要进行很多优化，例如在jdk的ConcurrentHashMap中就使用了这种方式，其中涉及元素尽量均匀、访问和操作速度要快、线程安全、扩容等很多问题。

我们来看一下下面这个Hash结构，下面的图有两处非常明显的错误，请你先想想是啥。



首先是数组的长度必须是2的n次幂，这里长度是9，明显有错，然后是entry 的个数不能大于数组长度的75%，如果大于就会触发扩容机制进行扩容，这里明显是大于75%， 正确的图应该是这样的：



数组的长度即是2的n次幂，而他的size又不大于数组长度的75%。 HashMap的实现原理是先要找到要存放数组的下标，如果是空的就存进去，如果不是空的就判断key值是否一样，如果一样就替换，如果不一样就以链表的形式存在链表中(从JDK8开始，根据元素数量选择使用链表还是红黑树存储)。

4. 高频面试题

4.1.栈实现队列和队列实现栈

栈的特点是后进先出，队的特点是先进先出。两个栈将底部拼接到一起就能实现队列的效果， 通过队列也能实现栈的功能。在很多地方能看到让你通过两个栈实现队列的题目，也有很多地方是两个队列实现栈的题目，我们就干脆一次看一下如何做。这正好对应LeetCode232和225两道题。

4.1.1 用栈实现队列

LeetCode232 请你仅使用两个栈实现先入先出队列。队列应当支持一般队列支持的所有操作（push、pop、peek、empty）：
实现 MyQueue 类：

```
void push(int x) 将元素 x 推到队列的末尾
int pop() 从队列的开头移除并返回元素
int peek() 返回队列开头的元素
boolean empty() 如果队列为空，返回 true ；否则，返回 false
```

这个题的思路是，将一个栈当作输入栈，用于压入 push 传入的数据；另一个栈当作输出栈，用于pop 和 peek 操作。

每次pop 或 peek 时，若输出栈为空则将输入栈的全部数据依次弹出并压入输出栈，这样输出栈从栈顶往栈底的顺序就是队列从队首往队尾的顺序。

代码难度不算大：

```
class MyQueue:
    def __init__(self):
        """
        in主要负责push，out主要负责pop
        """
        self.stack_in = []
        self.stack_out = []

    def push(self, x: int) -> None:
        """
        有新元素进来，就往in里面push
        """
        self.stack_in.append(x)

    def pop(self) -> int:
        """
        Removes the element from in front of queue and returns that element.
        """
```

```
        if self.empty():
            return None

        if self.stack_out:
            return self.stack_out.pop()
        else:
            for i in range(len(self.stack_in)):
                self.stack_out.append(self.stack_in.pop())
            return self.stack_out.pop()

    def peek(self) -> int:
        """
        Get the front element.
        """
        ans = self.pop()
        self.stack_out.append(ans)
        return ans

    def empty(self) -> bool:
        """
        只要in或者out有元素，说明队列不为空
        """
        return not (self.stack_in or self.stack_out)
```

这个还可以使用python的queue模块进一步简化操作：

```
import queue
class MyQueue:

    def __init__(self):
        self.q = queue.Queue(100)

    def push(self, x: int) -> None:
        self.q.put(x)

    def pop(self) -> int:
        return self.q.get()

    def peek(self) -> int:
        return self.q.queue[0]

    def empty(self) -> bool:
        return self.q.empty()
```

4.1.2 用队列实现栈

leetcode225 请你仅使用两个队列实现一个后入先出（LIFO）的栈，并支持普通栈的全部四种操作（push、top、pop 和 empty）。实现 MyStack 类：

```
void push(int x) 将元素 x 压入栈顶。
int pop() 移除并返回栈顶元素。
int top() 返回栈顶元素。
boolean empty() 如果栈是空的，返回 true ；否则，返回 false 。
```

分析：这个问题首先想到的是使用两个队列来实现。为了满足栈的特性，即最后入栈的元素最先出栈，在使用队列实现栈时，应满足队列前端的元素是最后入栈的元素。可以使用两个队列实现栈的操作，其中 queue1用于存储栈内的元素，queue2作为入栈操作的辅助队列。

入栈操作时，首先将元素入队到 queue2，然后将 queue1的全部元素依次出队并入队到queue2，此时 queue2的前端的元素即为新入栈的元素，再将 queue1和queue2互换，则 queue1的元素即为栈内的元素，queue 1的前端和后端分别对应栈顶和栈底。

由于每次入栈操作都确保queue1的前端元素为栈顶元素，因此出栈操作和获得栈顶元素操作都可以简单实现。出栈操作只需要移除queue1的前端元素并返回即可，获得栈顶元素操作只需要获得 queue 1的前端元素并返回即可（不移除元素）。

由于 queue 1用于存储栈内的元素，判断栈是否为空时，只需要判断 queue1是否为空即可。

```
class MyStack:

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.queue1 = collections.deque()
        self.queue2 = collections.deque()


    def push(self, x: int) -> None:
        """
        Push element x onto stack.
        """
        self.queue2.append(x)
        while self.queue1:
            self.queue2.append(self.queue1.popleft())
        self.queue1, self.queue2 = self.queue2, self.queue1

    def pop(self) -> int:
        """
        Removes the element on top of the stack and returns that element.
        """
        return self.queue1.popleft()
```

```
    def top(self) -> int:
        """
        Get the top element.
        """
        return self.queue1[0]

    def empty(self) -> bool:
        """
        Returns whether the stack is empty.
        """
        return not self.queue1
```

拓展这里还能用一个队列来实现，你能想到怎么做吗。

4.2 n数之和专题

很多人开始LeetCode的第一题就是求两数之和的问题，事实上除此之外，还有几个类似的问题，例如LeetCode15三数之和，LeetCode18.四数相加和 LeetCode454.四数相加II等等。我们就集中看一下。

4.2.1 两数之和

LeetCode1.给定一个整数数组 nums 和一个整数目标值 target，请你在该数组中找出 和为目标值 target 的那 两个 整数，并返回它们的数组下标。你可以假设每种输入只会对应一个答案。但是，数组中同一个元素在答案里不能重复出现。你可以按任意顺序返回答案。

```
示例1：
输入：nums = [2,7,11,15], target = 9
输出：[0,1]
解释：因为 nums[0] + nums[1] == 9 ，返回 [0, 1] 。

示例2：
输入：nums = [3,2,4], target = 6
输出：[1,2]
```

本题可以使用两层循环解决，第一层确定一个数，2，7，一直到11，然后内层循环继续遍历后序元素，判断是否存在target - x的数即可，代码如下：

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                if nums[i] + nums[j] == target:
                    return [i, j]

        return []
```

这种方式的不足在于寻找 `target - x` 的时间复杂度过高，我们可以使用哈希表，可以将寻找 target - x 的时间复杂度降低到从 O(N)降低到 O(1)。这样我们创建一个哈希表，对于每一个 x，我们首先查询哈希表中是否存在 target - x，然后将 x 插入到哈希表中，即可保证不会让 x 和自己匹配。

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        hashtable = dict()
        for i, num in enumerate(nums):
            if target - num in hashtable:
                return [hashtable[target - num], i]
            hashtable[nums[i]] = i
        return []
```

如果使用Hash还是比较容易的，时间复杂度低了，但是空间复杂度高了，那是否还有其他方法呢？这个真不多，不过假如告诉你原始数组是有序的那可以进一步优化，仍然采用两层循环的方式，外层仍然是一个个遍历，而内层循环可以换成二分，这样复杂度就从O(n^2)降低到O(nlogn)。感兴趣的同学可以尝试写一下看看。

4.2.2 三数之和

如果将两个数换成三个会怎样呢？LeetCode15.给你一个包含 n 个整数的数组 nums，判断 nums 中是否存在三个元素 a，b，c ，使得 a + b + c = 0 ？ 请你找出所有和为 0 且不重复的三元组。注意：答案中不可以包含重复的三元组。

```
示例1：
输入：nums = [-1,0,1,2,-1,-4]
输出：[[-1,-1,2],[-1,0,1]]
```

本题看似就是两数增加了一个数，但是难度增加了很多，我们可以使用三层循环直接找，时间复杂度为O(n^3)，太高了，放弃。也可以使用双层循环+Hash来实现，首先按照第一题两数之和的思路，我们可以固定一个数`traget`，再利用两数之和的思想去`map`中存取或查找`(-1)*target - num[j]`，但是这样的问题是无法消除重复结果，例如如果输入[-1,0,1,2,-1,-4]，返回的结果是[[-1,1,0],[-1,-1,2],[0,1,-1],[0,-1,1],[1,-1,0],[2,-1,-1]]，如果我们再增加一个去重方法，将直接导致执行超时。

我们就要想其他方法了，这个公认最好的方式是”排序+双指针“。我们可以先将数组排序来处理重复结果，然后还是固定一位元素，由于数组是排好序的，所以我们用双指针来不断寻找即可求解，代码如下：

```
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:

        n=len(nums)
        res=[]
        if(not nums or n<3):
            return []
        nums.sort()
        res=[]
        for i in range(n):
            if(nums[i]>0):
                return res
```

```
            if(i>0 and nums[i]==nums[i-1]):
                continue
            L=i+1
            R=n-1
            while(L<R):
                if(nums[i]+nums[L]+nums[R]==0):
                    res.append([nums[i],nums[L],nums[R]])
                    while(L<R and nums[L]==nums[L+1]):
                        L=L+1
                    while(L<R and nums[R]==nums[R-1]):
                        R=R-1
                    L=L+1
                    R=R-1
                elif(nums[i]+nums[L]+nums[R]>0):
                    R=R-1
                else:
                    L=L+1
        return res
```

拓展

如果我们继续拓展，在前面基础上再增加一个数呢？ 这就是LeetCode18.给你一个由 n 个整数组成的数组 nums ，和一个目标值 target 。请你找出并返回满足下述全部条件且不重复的四元组 [nums[a], nums[b], nums[c], nums[d]] ，满足nums[a] + nums[b] + nums[c] + nums[d] == target。 这个题最直接的想法是在上一题的基础上再套一层for循环来解决。 思路虽然简单，但是实现过程比较复杂，感兴趣的同学可以研究一下。

如果我们再拓展一下，如果四个数字不是在一个数组里，而是分别在四个数组里，让你从每个数组中分别获得一个元素，使得nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0，此时又该怎么做呢？这个就是LeetCode454题，感兴趣的同学可以研究一下。

4.3 括号匹配问题

首先看题目要求，LeetCode20. 给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串 s ，判断字符串是否有效。有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

```
示例1：
输入：s = "()[]{}"
输出：true
```

本题还是比较简单的，其中比较麻烦的是如何判断两个符号是不是一组的，我们可以用哈希表将所有符号先存储，左半边做`key`,右半边做`value`。遍历字符串的时候，遇到左半边符号就入栈，遇到右半边符号就与栈顶的符号比较，不匹配就返回false

```
class Solution:
    def isValid(self, s: str) -> bool:
        if len(s) % 2 == 1:
```

```
            return False

        pairs = {
            ")": "(",
            "]": "[",
            "}": "{",
        }
        stack = list()
        for ch in s:
            if ch in pairs:
                if not stack or stack[-1] != pairs[ch]:
                    return False
                stack.pop()
            else:
                stack.append(ch)

        return not stack
```

我们也可以使用更简单的方式来实现：

```
class Solution:
  def isValid(self, s: str) -> bool:
      dic = {'{': '}',  '[': ']', '(': ')', '?': '?'}
      stack = ['?']
      for c in s:
          if c in dic: stack.append(c)
          elif dic[stack.pop()] != c: return False
      return len(stack) == 1
```

LeetCode给我们造了十几个括号匹配的问题，都是条件变来变去，但是解决起来有难有易，如果你感兴趣，可以继续研究一下：

LeetCode22 .括号生成

LeetCode32.最长有效括号

LeetCode301.删除无效的括号

LeetCode687. 有效的括号字符串

LeetCode856.括号的分数

LeetCode1087.花括号展开

LeetCode1111.有效括号的嵌套深度

4.3 最大栈和最小栈问题

4.3.1 最小栈

LeetCode 155，设计一个支持 push ， pop ， top 操作，并能在常数时间内检索到最小元素的栈。

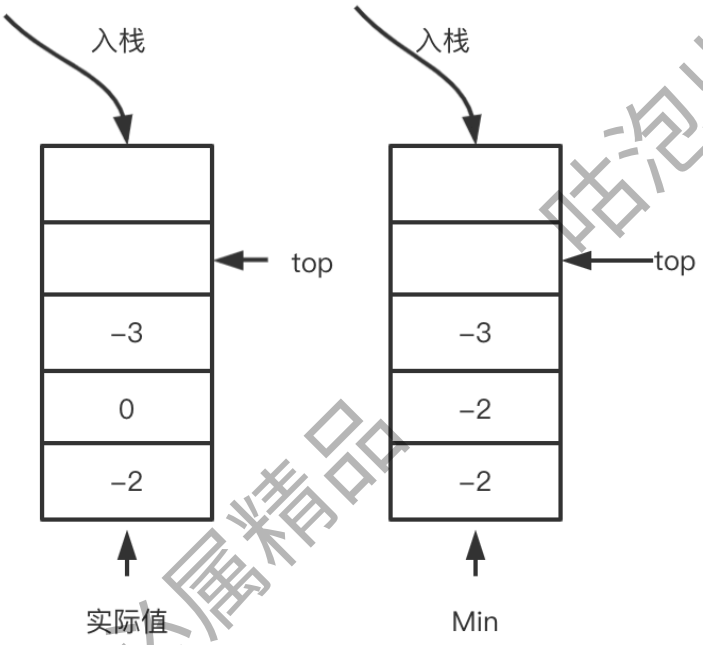
实现 MinStack 类:

```
MinStack() 初始化堆栈对象。
void push(int val) 将元素val推入堆栈。
void pop() 删除堆栈顶部的元素。
int top() 获取堆栈顶部的元素。
int getMin() 获取堆栈中的最小元素。
```

示例：

```
输入：
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

输出：
[null,null,null,null,-3,null,0,-2]

解释：
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin();   --> 返回 -3.
minStack.pop();
minStack.top();      --> 返回 0.
minStack.getMin();   --> 返回 -2.
```

本题的关键在于理解getMin()到底表示什么，可以看一个例子上面的示例画成示意图如下：



这里的关键是理解对应的Min栈内，中间元素为什么是-2，理解了本题就非常简单。

题目要求在常数时间内获得栈中的最小值，因此不能在 getMin() 的时候再去计算最小值，最好应该在 push 或者 pop 的时候就已经计算好了当前栈中的最小值。

对于栈来说，如果一个元素 a 在入栈时，栈里有其它的元素 b, c, d，那么无论这个栈在之后经历了什么操作，只要 a 在栈中，b, c, d 就一定在栈中，因为在 a 被弹出之前，b, c, d 不会被弹出。

因此，在操作过程中的任意一个时刻，只要栈顶的元素是 a，那么我们就可以确定栈里面现在的元素一定是 a, b, c, d。

那么，我们可以在每个元素 a 入栈时把当前栈的最小值 m 存储起来。在这之后无论何时，如果栈顶元素是 a，我们就可以直接返回存储的最小值 m。

按照上面的思路，我们只需要设计一个数据结构，使得每个元素 a 与其相应的最小值 m 时刻保持一一对应。因此我们可以使用一个辅助栈，与元素栈同步插入与删除，用于存储与每个元素对应的最小值。

当一个元素要入栈时，我们取当前辅助栈的栈顶存储的最小值，与当前元素比较得出最小值，将这个最小值插入辅助栈中；

当一个元素要出栈时，我们把辅助栈的栈顶元素也一并弹出；

在任意一个时刻，栈内元素的最小值就存储在辅助栈的栈顶元素中。

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min_stack = [math.inf]

    def push(self, x: int) -> None:
        self.stack.append(x)
        self.min_stack.append(min(x, self.min_stack[-1]))
```

```
    def pop(self) -> None:
        self.stack.pop()
        self.min_stack.pop()

    def top(self) -> int:
        return self.stack[-1]

    def getMin(self) -> int:
        return self.min_stack[-1]
```

4.3.2 LeetCode 716 最大栈

LeetCode 716.设计一个最大栈数据结构，既支持栈操作，又支持查找栈中最大元素。

实现 MaxStack 类：

```
MaxStack() 初始化栈对象
void push(int x) 将元素 x 压入栈中。
int pop() 移除栈顶元素并返回这个元素。
int top() 返回栈顶元素，无需移除。
int peekMax() 检索并返回栈中最大元素，无需移除。
int popMax() 检索并返回栈中最大元素，并将其移除。如果有多个最大元素，只要移除 最靠近栈顶 的那个。
```

示例：

```
输入
["MaxStack", "push", "push", "push", "top", "popMax", "top", "peekMax", "pop", "top"]
[[], [5], [1], [5], [], [], [], [], [], []]
输出
[null, null, null, null, 5, 5, 1, 5, 1, 5]

解释
MaxStack stk = new MaxStack();
stk.push(5);   // [5] - 5 既是栈顶元素，也是最大元素
stk.push(1);   // [5, 1] - 栈顶元素是 1，最大元素是 5
stk.push(5);   // [5, 1, 5] - 5 既是栈顶元素，也是最大元素
stk.top();     // 返回 5，[5, 1, 5] - 栈没有改变
stk.popMax();  // 返回 5，[5, 1] - 栈发生改变，栈顶元素不再是最大元素
stk.top();     // 返回 1，[5, 1] - 栈没有改变
stk.peekMax(); // 返回 5，[5, 1] - 栈没有改变
stk.pop();     // 返回 1，[5] - 此操作后，5 既是栈顶元素，也是最大元素
stk.top();     // 返回 5，[5] - 栈没有改变
```

本题与上一题的相反，但是处理方法是一致的。一个普通的栈可以支持前三种操作 push(x)，pop() 和 top()，所以我们需要考虑的仅为后两种操作 peekMax() 和 popMax()。

对于 peekMax()，我们可以另一个栈来存储每个位置到栈底的所有元素的最大值。例如，如果当前第一个栈中的元素为 [2, 1, 5, 3, 9]，那么第二个栈中的元素为 [2, 2, 5, 5, 9]。在 push(x) 操作时，只需要将第二个栈的栈顶和 xx 的最大值入栈，而在 pop() 操作时，只需要将第二个栈进行出栈。

对于 popMax()，由于我们知道当前栈中最大的元素值，因此可以直接将两个栈同时出栈，并存储第一个栈出栈的所有值。当某个时刻，第一个栈的出栈元素等于当前栈中最大的元素值时，就找到了最大的元素。此时我们将之前出第一个栈的所有元素重新入栈，并同步更新第二个栈，就完成了 popMax() 操作。

```
class MaxStack(list):
    def push(self, x):
        m = max(x, self[-1][1] if self else None)
        self.append((x, m))

    def pop(self):
        return list.pop(self)[0]

    def top(self):
        return self[-1][0]

    def peekMax(self):
        return self[-1][1]

    def popMax(self):
        m = self[-1][1]
        b = []
        while self[-1][0] != m:
            b.append(self.pop())

        self.pop()
        map(self.push, reversed(b))
        return m
```