#### [136. 只出现一次的数字](https://leetcode-cn.com/problems/single-number/)

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗？

**方法1.哈希表：**

class Solution {

public int singleNumber(int[] nums) {

Map<Integer,Integer> hash=new HashMap<>();

int count=0;

for(int i=0;i<nums.length;i++)

{

if(hash.get(nums[i])==null)

{

count=1;

hash.put(nums[i],count);//第一个是key，第二个是value

}

else if(count>0)

{

count++;

hash.put(nums[i],count);

}

}

for(int i=0;i<nums.length;i++)

{

if(hash.get(nums[i])==1)

{

return nums[i];

}

}

return -1;

}

}

**方法2.异或运算：**

异或解法：异或运算满足交换律，a^b^a=a^a^b=b,因此ans相当于nums[0]^nums[1]^nums[2]^nums[3]^nums[4]..... 然后再根据交换律把相等的合并到一块儿进行异或（结果为0），然后再与只出现过一次的元素进行异或，这样最后的结果就是，只出现过一次的元素（0^任意值=任意值）

class Solution {

public int singleNumber(int[] nums) {

int ans=nums[0];

if(nums.length>1)

{

for(int i=1;i<nums.length;i++)

{

ans=ans^nums[i];

}

}

return ans;

}

}

注：这里关于哈希表的put这个存放的值是(key,value),通过get我们可以获得key值对应的value值

1. **环形链表**

给定一个链表，判断链表中是否有环。为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

方法1.hashset:

HashSet简单的理解就是HashSet对象中不能存储相同的数据，存储数据时是无序的。但是HashSet存储元素的顺序并不是按照存入时的顺序（和List显然不同） 是按照哈希值来存的所以取数据也是按照哈希值取得。存储是无序的这就和C++里的Set就不一样了C++里面的Set是有序的我认为这是在使用时候的主要区别

public boolean hasCycle(ListNode head) {

Set<ListNode> nodesSeen = new HashSet<>();

while (head != null) {

if (nodesSeen.contains(head)) {

return true;//如果在哈希表中已经包括过这个元素，那么说明存在闭环，返回真值

} else {

nodesSeen.add(head);//在表中没有出现过这个头结点，因此将这个头结点放入哈希表中

}

head = head.next;

}

return false;

}

方法2.双指针：

通过使用具有不同速度的快、慢两个指针遍历链表，空间复杂度可以被降低至 O(1)O(1)。慢指针每次移动一步，而快指针每次移动两步。

如果列表中不存在环，最终快指针将会最先到达尾部，此时我们可以返回 false。

现在考虑一个环形链表，把慢指针和快指针想象成两个在环形赛道上跑步的运动员（分别称之为慢跑者与快跑者）。而快跑者最终一定会追上慢跑者。这是为什么呢？考虑下面这种情况（记作情况 A）- 假如快跑者只落后慢跑者一步，在下一次迭代中，它们就会分别跑了一步或两步并相遇。

其他情况又会怎样呢？例如，我们没有考虑快跑者在慢跑者之后两步或三步的情况。但其实不难想到，因为在下一次或者下下次迭代后，又会变成上面提到的情况 A。

public class Solution {

public boolean hasCycle(ListNode head) {

if(head==null||head.next==null)

return false;//如果链表中没有元素或者链表中只有一个元素

ListNode slow=head;

ListNode fast=head.next;

while(slow!=fast)//如果存在闭环，快指针一定会追上慢指针

{

if(fast==null||fast.next==null)//如果是奇数个个数，比如3 2 0，那么进行一次循环后，fast=null，如果是偶数个，3 2 0 4，那么一次循环后，fast.next=null

{

return false;

}

slow=slow.next;

fast=fast.next.next;//让快指针一次走两步

}

return true;

}

}

1. **最小栈**

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。

push(x) -- 将元素 x 推入栈中。

pop() -- 删除栈顶的元素。

top() -- 获取栈顶元素。

getMin() -- 检索栈中的最小元素。

**方法1.辅助栈和数据栈同步**

class MinStack {

private Stack<Integer> data;

private Stack<Integer> helper;//辅助栈用来存储最小值，及他的top是最小值

/\*\* initialize your data structure here. \*/

public MinStack() {

data=new Stack<>();

helper=new Stack<>();

}

public void push(int x) {

data.add(x);

if(helper.isEmpty()||helper.peek()>=x)//如果辅助栈是空的，或者他的top值比xda，那么需要把xpush到辅助栈中

{

helper.add(x);

}

else

{

helper.add(helper.peek());

}

}

public void pop() {

if(!data.isEmpty())

{

helper.pop();//在push时，两者是保持同步的，因此最小值的原始值存入地也是相同的

data.pop();

}

}

public int top() {

if(!data.isEmpty())

{

return data.peek();

}

throw new RuntimeException("栈中元素为空，此操作非法");

}

public int getMin() {

if(!helper.isEmpty())

{

return helper.peek();//因为在辅助栈中存储的始终是最小的元素

}

throw new RuntimeException("栈中元素为空，此操作非法");

}

}

//异常处理方法： throw new RuntimeException("栈中元素为空，此操作非法");

**方法2.辅助栈和数据栈不同步的情况：**

public class MinStack {

// 数据栈

private Stack<Integer> data;

// 辅助栈

private Stack<Integer> helper;

/\*\*

\* initialize your data structure here.

\*/

public MinStack() {

data = new Stack<>();

helper = new Stack<>();

}

// 思路 2：辅助栈和数据栈不同步

// 关键 1：辅助栈的元素空的时候，必须放入新进来的数

// 关键 2：新来的数小于或者等于辅助栈栈顶元素的时候，才放入（特别注意这里等于要考虑进去）

// 关键 3：出栈的时候，辅助栈的栈顶元素等于数据栈的栈顶元素，才出栈，即"出栈保持同步"就可以了

public void push(int x) {

// 辅助栈在必要的时候才增加

data.add(x);

// 关键 1 和 关键 2

if (helper.isEmpty() || helper.peek() >= x) {

helper.add(x);//这里元素只放入一次

}

}

public void pop() {

// 关键 3：data 一定得 pop()

if (!data.isEmpty()) {

// 注意：声明成 int 类型，这里完成了自动拆箱，从 Integer 转成了 int，因此下面的比较可以使用 "==" 运算符

// 参考资料：https://www.cnblogs.com/GuoYaxiang/p/6931264.html

// 如果把 top 变量声明成 Integer 类型，下面的比较就得使用 equals 方法

int top = data.pop();

if(top == helper.peek()){

//在这里，只有当辅助栈和数据栈的元素相同时，才会弹出辅助栈的元素

helper.pop();

}

}

}

public int top() {

if(!data.isEmpty()){

return data.peek();

}

throw new RuntimeException("栈中元素为空，此操作非法");

}

public int getMin() {

if(!helper.isEmpty()){

return helper.peek();

}

throw new RuntimeException("栈中元素为空，此操作非法");

}

}

注：声明成 int 类型，这里完成了自动拆箱，从 Integer 转成了 int，因此下面的比较可以使用 "==" 运算符

参考资料：https://www.cnblogs.com/GuoYaxiang/p/6931264.html

如果把 top 变量声明成 Integer 类型，下面的比较就得使用 equals 方法

1. **相交链表**

编写一个程序，找到两个单链表相交的起始节点。

很聪明的算法：指针 pA 指向 A 链表，指针 pB 指向 B 链表，依次往后遍历

如果 pA 到了末尾，则 pA = headB 继续遍历

如果 pB 到了末尾，则 pB = headA 继续遍历

比较长的链表指针指向较短链表head时，长度差就消除了

如此，只需要将最短链表遍历两次即可找到位置

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA == null || headB == null) return null;

ListNode pA = headA, pB = headB;

while (pA != pB) {

pA = pA == null ? headB : pA.next;

pB = pB == null ? headA : pB.next;

}

return pA;

}

**一样的思路，自己写的，包括其中错误部分的解析**

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

if (headA == null || headB == null) return null;

ListNode pa=headA;

ListNode pb=headB;

while(pa!=pb)

{

//pa=pa.next;如果这样会陷入死循环，因为如果pa和pb的下一个都为null，那么就会进行后面两个if中的运算

//pb=pb.next;

//if(pa==null&&pb==null) return null;

if(pa==null)

pa=headB;

else

pa=pa.next;

if(pb==null)

pb=headA;

else

pb=pb.next;

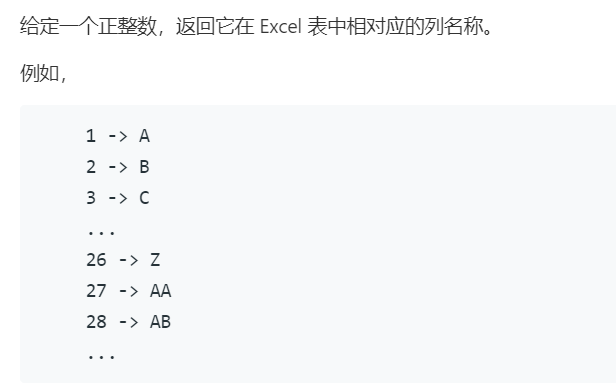
}

return pa;

}

}

**168. Excel表列名称**



1.stringbulider中的insert方法：

java API解释：将 char 参数的字符串表示形式插入此序列中。

通俗点说，就是在原序列的offset处插入字符char并生成新的StringBuider对象。

举例：

StringBuilder strb=new StringBuilder("123");

System.out.println(strb);//输出：123

strb.insert(1, 'c');

System.out.println(strb);//输出：1c23

代码：

public String convertToTitle(int n) {

StringBuilder sb = new StringBuilder();

while (n > 0) {

int c = n % 26;

if(c == 0){

c = 26;

**n -= 1;//关键的一步,比如Z,经过这一部运算后，n=25，因此循环结束**

}

sb.insert(0, (char) ('A' + c - 1));

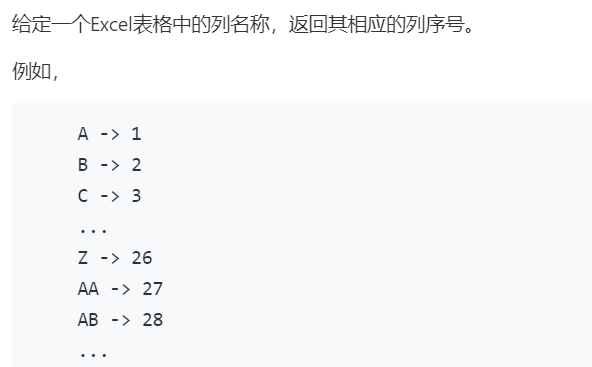
n /= 26;

}

return sb.toString();

}

1. **Excel表列序号**



class Solution {

public int titleToNumber(String s) {

int ans = 0;

for(int i=0;i<s.length();i++) {

int num = s.charAt(i) - 'A' + 1;

**ans = ans \* 26 + num;//AA可以看成1\*26+1，AB可以看成1\*26+2，这里的1是前一次运算中ans中存储的值**

}

return ans;

}

}