# 数组



## 搜索插入位置（35题）

暴力解法：通过循环判断是否大于或等于目标值，如果满足条件则返回下标。如果不满足返回数组长度。缺点：时间消耗过高。

for(let i=0;i<length;i++){

if(nums[i]>=target){

return i

}

}

Return len

二分法：通过取中间值分为左右区间进行判断，注意中间值取值的问题。/2和>>1有区别

思路：通过取中间值，来快速判断目标值得范围。将会有几个情况的出现

1. 目标值在数组所有元素之前[0,-1]
2. 目标值等于数组中某一个元素 return mid
3. 目标值在数组元素之间[left,right],right取值为n-1,while(left <= right) return right+1。如果目标值在[left,right)之间，right取值为n。则while(left < right)返回return right。？
4. 目标值在所有元素之后 return right+1

注意：中位数取值的问题，找出mid，该索引为 mid =（left + right）/ 2，但是这样写有可能溢出，所以我们需要改进一下写成mid = left +（right - left）/ 2 或者 left + ((right - left ) >> 1) 两者作用是一样的，都是为了找到两指针的中间索引，使用位运算的速度更快。

代码：

var searchInsert = function(nums, target) {

   let left=0;

  let right=nums.length-1;//定义目标值的范围在[left,right]

  while(left<=right){

      const middle=(left+right)>>1;//注意这个判断,直接/2需要用parseINT转换，不然会报错

      if(nums[middle]==target){return middle;}

      else if(nums[middle]>target){

          right=middle-1;

      }else{

          left=middle+1;

      }

  }

  return left;

};

## 寻找旋转排序数组中的最小值（153题）

思路：左右不对称边界。用二分法将目标值锁定，并不断收缩边界。

1. 左<中<右，说明没有旋转，最小值在左边
2. 左>中，右>中，说明有旋转，最小值在左半边，可以收缩右边界。
3. 左<中，中>右，说明有旋转，最小值在右半边，收缩左边界。
4. 左>中>右。不会出现。

如果用中值和左值比较，则1,3,两个情况无法区分，所以用中值和右值进行比较。边界区间为左闭右开[left,right）,因为当left==right值时，数组只有一位值，直接输出。

代码：

var findMin = function(nums) {

    let left=0,right=nums.length-1;

    if(left===right) return nums[left];

    if(nums[left]<nums[right]) return nums[left];

    while(left<right){

        let mid=(left+right)>>1;

        //如果中位值大于最右边界值，说明翻转无序区域在右边,收敛左边界

        if(nums[mid]>nums[right]){

            left=mid+1;

        }else{

            right=mid;

        }

    }

    return nums[left];

};

## 第一个错误的版本（278题）

注意:还是中位数取值的问题

Let mid=(left+right)>>1 出错应该用>>>1

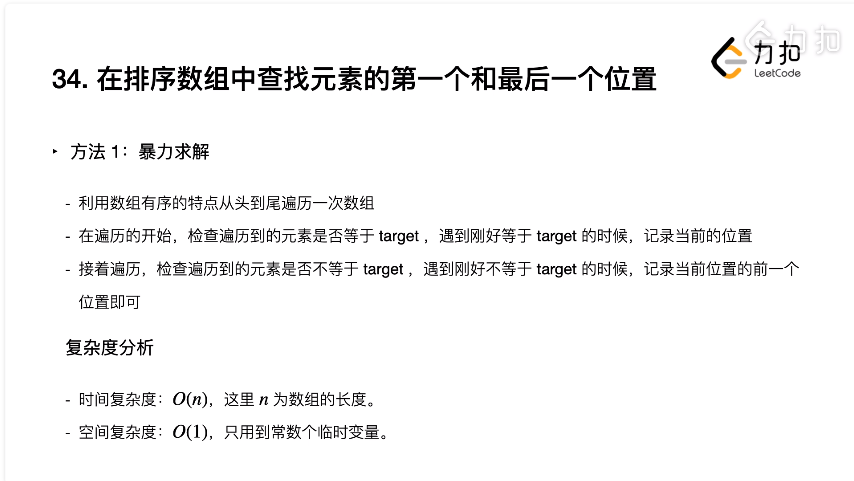
>>表示是带符号的右移bai：按照du二进制把数字右移指定数位，zhi符号dao位为正补零，符号位负补zhuan一，低位直接移除。

>>>表示shu无符号的右移：按照二进制把数字右移指定数位，高位直接补零，低位移除。

区别：>>在传递时也把符号一起传递，比如+3、-2在传递再传出时依然是+3、-2，而使用>>>时就会统一变为3、2。带符号于无符号的差别就在此。

## 在排序数组中查找元素的第一个和最后一个位置(34题)

思路1：暴力解决。



思路二：二分法。通过查找第一个大于或等于mid的数据为left,和第一个大于mid的数据-1为right.分为两个查找过程进行查找。考虑特殊情况，一是数组为空返回[-1,-1]；而是二分查找过程中没有这个值,此时也返回【-1,-1】。

代码：

var searchRange = function(nums, target) {

var len=nums.length;

//特殊情况一，数组为空

if(len==0) return [-1,-1];

//求取第一个数值

    var leftRange=function(){

        let left=0;

        let right=len-1;

        while(left<right){

            let mid=(left+right)>>1;

            if(nums[mid]<target) {

//当中位数的值小于target时，此时范围为[mid+1,right]

                left=mid+1;

            }else if(nums[mid]==target){

//当中位数值和目标值相等时，可能出现左右两边都有的情况，此时收敛右区间来查找左边。范围为[left,mid];

                right=mid;

            }else if(nums[mid]>target){

//当中位值大于目标值时，范围在[left,mid-1]

                right=mid-1;

            }

        }

//特殊情况二，数组中没有这个元素，返回[-1,-1]

        if(nums[left]!=target) return -1;

        return left;

}

//求取最后一个元素出现的位置，就是查找第一个不等于target值得位置-1

      var rightRange=function(){

        let left=0;

        let right=len-1;

        while(left<right){

            let mid=(left+right+1)>>1;

            if(nums[mid]<target) {

                //mid<target时,target的范围在[mid+1,right]

                left=mid+1;

            }else if(nums[mid]==target){

//target的范围为[mid+1,right]

                left=mid;

            }else if(nums[mid]>target){

//target的范围为[left,mid-1]

                right=mid-1;

            }

        }

        return left;

    }

    let left=leftRange();

    let right=rightRange();

    if(left==-1){

        return [-1,-1];

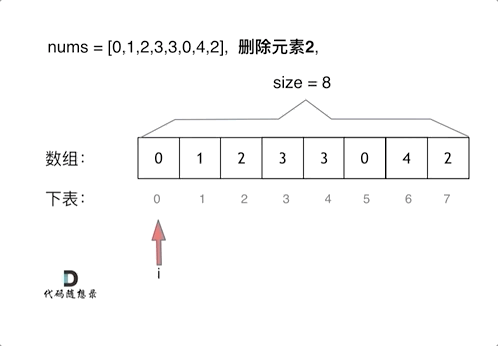
    }

    return [left,right]

};

## 移除元素（27题）

暴力解法：通过循环查找目标元素，然后在通过内层循环替换数组元素。



代码：

var removeElement = function(nums, val) {

    var len=nums.length;

    for(let i=0;i<len;i++){

        if(nums[i]==val){

            for(let j=i+1;j<len;j++){

                nums[j-1]=nums[j]

            }

            i--;

            len--;

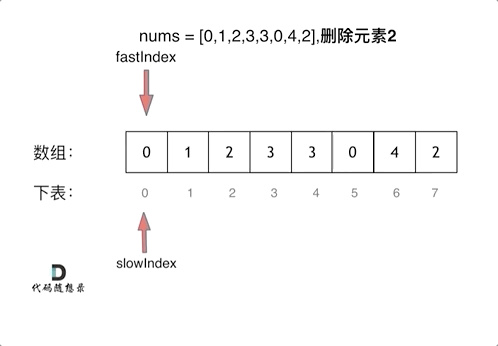
        }

    }

return len;

};

双指针法（快慢指针法）：「通过一个快指针和慢指针在一个for循环下完成两个for循环的工作。」



代码：

var removeElement = function(nums, val) {

    let slow=0;

    for(let fast=0;fast<nums.length;fast++){

        if(nums[fast]!=val){

            nums[slow++]=nums[fast];

        }

    }

    return slow;

};

## 删除排序数组中的重复项（26题）

思路：双指针进行判断，如果不相同，则覆盖慢指针后一项内容

代码：

var removeDuplicates = function(nums) {

    let i=0,j;

    for(j=i+1;j<nums.length;j++){

        if(nums[i]!=nums[j]){

            i++

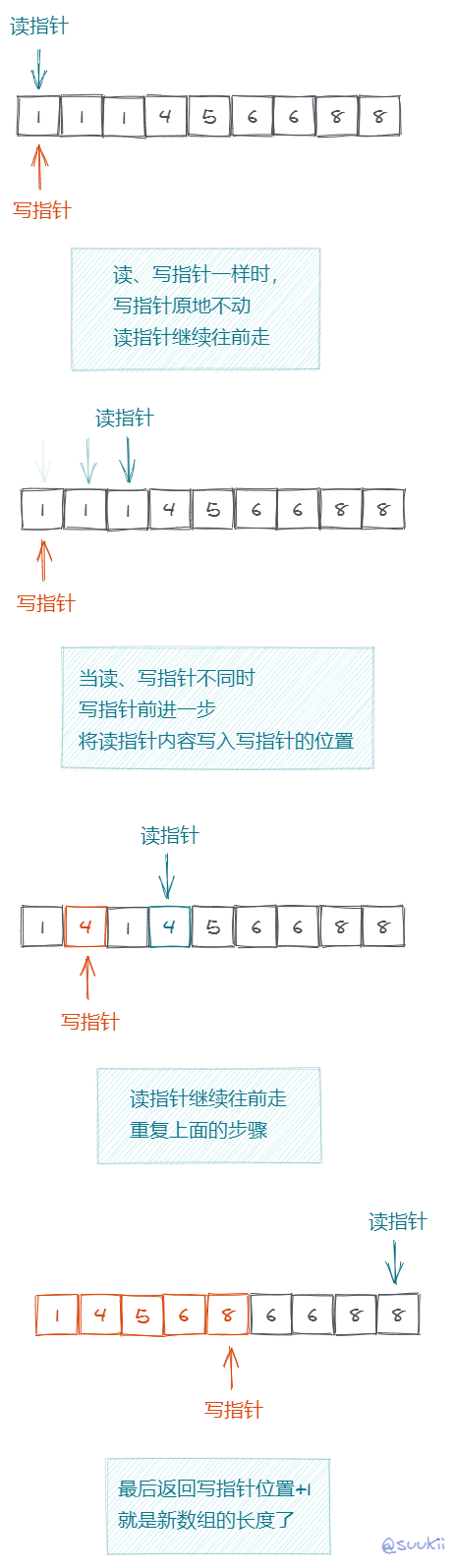
        nums[i]=nums[j]

        }

    }

    return i+1

};



## 长度最短的子数组（209题）

暴力解法：双重循环

代码：

var minSubArrayLen = function(s, nums) {

    let res=Infinity;

    let len=nums.length;

    for(let i=0;i<len;i++){

        let sum=0;

        for(let j=i;j<len;j++){

            sum+=nums[j];

            if(sum>=s)

            {

                res=Math.min(res,j-i+1)

            }

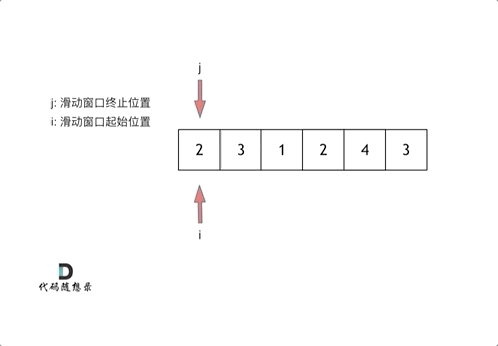
        }

    }

    return res==Infinity?0:res;

};

滑动窗口：**不断的调节子序列的起始位置和终止位置**



代码：

var minSubArrayLen = function(s, nums) {

    let res=Infinity;

    let len=nums.length;

    var i=0,sum=0;

    for(let j=0;j<len;j++)

    {

        sum+=nums[j];

        while(sum>=s){

            res=Math.min(res,j-i+1);

            sum-=nums[i++];//这里体现出滑动窗口的精髓之处，不断变更i（子序列的起始位置）

        }

    }

    return res==Infinity?0:res;

};

## 螺旋矩阵II（59题）

思路：边界问题，向内收敛

模拟填充过程：

声明一个 n\*n 的矩阵

填充未填充部分的首行（更新边界 rowStart++）

填充未填充部分的尾列（更新边界 columnEnd--）

填充未填充部分的尾行（更新边界 rowEnd--）

填充未填充部分的首列（更新边界 columnStart++）

重复上面步骤直到 num 递增到 n\*n

Ps：

创建数组

利用new Array()创建二维数组

var a = new Array();

for(var i=0;i<n;i++)

a[i] = new Array();

代码：

var generateMatrix = function(n) {

    let arr = new Array(n);

    for(let k = 0; k < n; k++){

        arr[k] = new Array(n);

    }

    let num=1,rowStart=0,rowEnd=n-1,colStart=0,colEnd=n-1;

    while(num<=n\*n){

        for(let i=colStart;i<=colEnd;i++){

            arr[rowStart][i]=num++;

        }

        rowStart++;

        for(let i=rowStart;i<=rowEnd;i++){

            arr[i][colEnd]=num++;

        }

        colEnd--;

        for(let i=colEnd;i>=colStart;i--){

            arr[rowEnd][i]=num++;

        }

        rowEnd--;

        for(let i=rowEnd;i>=rowStart;i--){

            arr[i][colStart]=num++

        }

        colStart++;

    }

    return arr

};

## 螺旋矩阵（54题）

思路：不停更新边界，并向内收敛。

（top,left）→ (top,right)

↑ ↓

(bottom,left)←(bottom,right)

代码：

var spiralOrder = function(matrix) {

const res = []

    //if(matrix==null||matrix.length==0) return res;

    let left=0, right=matrix[0].length-1, top=0, bottom=matrix.length-1;

    let sum=matrix[0].length\*matrix.length;

    while(res.length!==sum)

    {

        //从左到右

        for(let i=left;i<=right&&res.length<sum;i++){

            res.push(matrix[top][i]);

        }

        top++;

        //从上到下

        for(let i=top;i<=bottom&&res.length<sum;i++){

            res.push(matrix[i][right]);

        }

        right--;

        //从右到左

        for(let i=right;i>=left&&res.length<sum;i--){

            res.push(matrix[bottom][i])

        }

        bottom--;

        //从下到上

        for(let i=bottom;i>=top&&res.length<sum;i--)

        {

            res.push(matrix[i][left])

        }

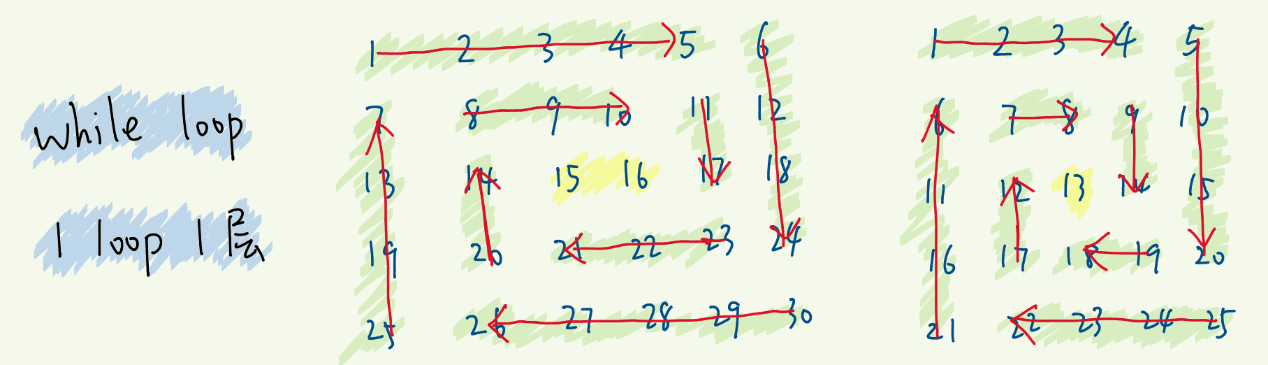
        left++;

    }

return res

};

注意：循环结束的条件需要注意。



思路概括

* 一层层向里处理，按顺时针依次遍历：上、右、下、左层
* 不再形成“环”了，就会剩下一行或一列，然后单独判断

代码：

var spiralOrder = function (matrix) {

if (matrix.length === 0) return []

const res = []

let top = 0, bottom = matrix.length - 1, left = 0, right = matrix[0].length - 1

while (top < bottom && left < right) {

for (let i = left; i < right; i++) res.push(matrix[top][i]) // 上层

for (let i = top; i < bottom; i++) res.push(matrix[i][right]) // 右层

for (let i = right; i > left; i--) res.push(matrix[bottom][i])// 下层

for (let i = bottom; i > top; i--) res.push(matrix[i][left]) // 左层

right--

top++

bottom--

left++ // 四个边界同时收缩，进入内层

}

if (top === bottom) // 剩下一行，从左到右依次添加

for (let i = left; i <= right; i++) res.push(matrix[top][i])

else if (left === right) // 剩下一列，从上到下依次添加

for (let i = top; i <= bottom; i++) res.push(matrix[i][left])

return res

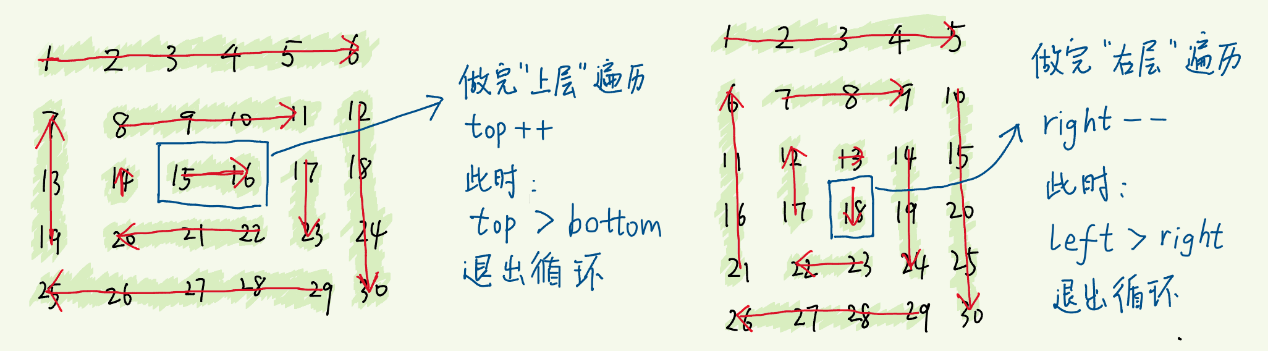
};

思路：遍历到底

* 能出现 循环途中，循环的条件不再满足
* 即出现 top > bottom || left > right ，其中一对边界彼此交错了
* 说明此时所有项都遍历完了，如果没有 break ，就会重复遍历

解决办法

* 每遍历完一条边，更新完相应的边界后，都加一条 if (top > bottom || left > right) break
* 但你发现，遍历完成要么发生在遍历完“上边”，要么发生在遍历完“右边”
* 所以只需在这两步操作之后，加 if (top > bottom || left > right) break 即可



[[1]](#endnote-1)[[2]](#endnote-2)

代码：

var spiralOrder = function (matrix) {

if (matrix.length == 0) return []

const res = []

let top = 0, bottom = matrix.length - 1, left = 0, right = matrix[0].length - 1

while (top <= bottom && left <= right) {

for (let i = left; i <= right; i++) res.push(matrix[top][i])

top++

for (let i = top; i <= bottom; i++) res.push(matrix[i][right])

right--

if (top > bottom || left > right) break

for (let i = right; i >= left; i--) res.push(matrix[bottom][i])

bottom--

for (let i = bottom; i >= top; i--) res.push(matrix[i][left])

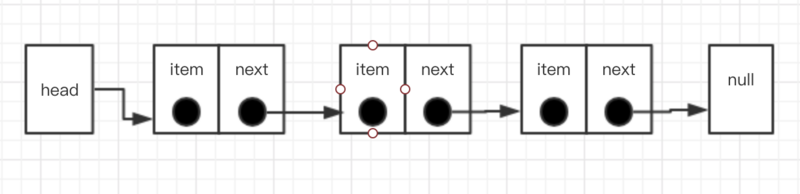
left++

}

return res

};

# 链表



学习网页：<https://www.cnblogs.com/jaxu/p/11277732.html>

## 移除链表元素（203题）

思路1：直接在原来链表进行操作。考虑两个情况，目标元素在头部时和目标元素不为头部元素。如果目标元素是头部元素，将头结点往后移。

代码：

var removeElements = function(head, val) {

    //头结点为空

    if(head==null) return null;

    //头结点不为空且头结点为目标元素值

   while(head!=null&&head.val===val){

       head=head.next;

   }

   //目标元素在后边

   var prev=head;

   while(prev.next!=null){

       if(prev.next.val==val){

           prev.next=prev.next.next;

       }else{

           prev=prev.next;

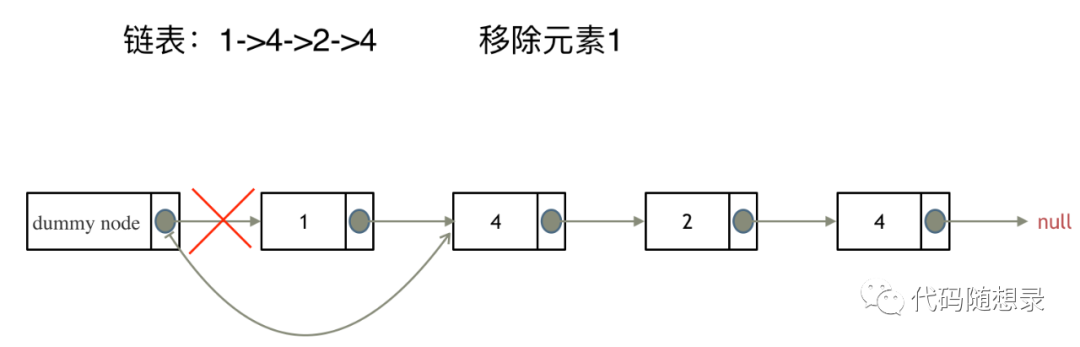
       }

   }

   return head;

};

思路二：使用虚拟头结点。注意return 头结点的时候，别忘了 return dummyNode->next;， 这才是新的头结点[[3]](#endnote-3)



代码：

var removeElements = function(head, val) {

    //创建虚拟头结点

    var dummy=new ListNode();

    dummy.next=head;

    var prev=dummy;

    while(prev.next!=null){

        if(prev.next.val===val){

            prev.next=prev.next.next;

        }else{

            prev=prev.next;

        }

    }

    return dummy.next;

};

思路三：递归

代码：

var removeElements = function(head, val) {

   //如果头结点为空

   if(head==null) return null;

   head.next=removeElements(head.next,val);

   if(head.val==val){

       return head.next;

   }else{

       return head;

   }

};

## 设计链表（707题）

* 获取链表第index个节点的数值
* 在链表的最前面插入一个节点
* 在链表的最后面插入一个节点
* 在链表第index个节点前面插入一个节点
* 删除链表的第index个节点

思路一：单链表

代码：

//链表类

var MyLinkedList = function() {

    this.len=0;

    this.head=null;

};

//辅助列描述链表中节点

var ListNode=function(val){

    this.val=val;

    this.next=null;

}

/\*\*

 \* Get the value of the index-th node in the linked list. If the index is invalid, return -1.

 \* @param {number} index

 \* @return {number}

 \*/

MyLinkedList.prototype.get = function(index) {

    //如果链表为空，返回-1

    if(this.len===0) return -1;

    //判断索引index的边界，如果值超出了范围（小于0或者>len-1）则无效返回-1;

    if(index<0||index>=this.len) return -1;

    //链表的索引从head->next开始

    let cur=this.head;

    for(let i=0;i<index;i++){

        cur=cur.next;

    }

    return cur.val;

};

MyLinkedList.prototype.getItem = function(index) {

    //如果链表为空，返回-1

    if(this.len===0) return -1;

    //判断索引index的边界，如果值超出了范围（小于0或者>len-1）则无效返回-1;

    if(index<0||index>=this.len) return -1;

    //链表的索引从head->next开始

    let cur=this.head;

    for(let i=0;i<index;i++){

        cur=cur.next;

    }

    return cur;

};

/\*\*

 \* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.

 \* @param {number} val

 \* @return {void}

 \*/

MyLinkedList.prototype.addAtHead = function(val) {

  //创建新节点

    const dummy=new ListNode(val);

    dummy.next=this.head;

    this.head=dummy;

    this.len++;

};

/\*\*

 \* Append a node of value val to the last element of the linked list.

 \* @param {number} val

 \* @return {void}

 \*/

MyLinkedList.prototype.addAtTail = function(val) {

     const dummy=new ListNode(val);

     //如果链表为空，则head指向当前元素

     if(this.head===null) this.head=dummy;

     else{

         //找到链表尾部的元素，然后添加新元素

         const cur=this.getItem(this.len-1);

         cur.next=dummy;

     }

     this.len++;

};

/\*\*

 \* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted.

 \* @param {number} index

 \* @param {number} val

 \* @return {void}

 \*/

MyLinkedList.prototype.addAtIndex = function(index, val) {

  if(index>this.len) return;

  if(index<=0)  return this.addAtHead(val);

  if(index==this.len) return this.addAtTail(val);

  const prev=this.getItem(index-1);

  const cur=new ListNode(val);

  cur.next=prev.next;

  prev.next=cur;

  this.len++;

};

/\*\*

 \* Delete the index-th node in the linked list, if the index is valid.

 \* @param {number} index

 \* @return {void}

 \*/

MyLinkedList.prototype.deleteAtIndex = function(index) {

    if(this.len===0) return;

     if (index < 0 || index >= this.len) return;

     if(index===0){

         this.head=this.head.next;

     }

     if(index>0&&index<this.len){

         const prev=this.getItem(index-1);

         const next=this.getItem(index+1);

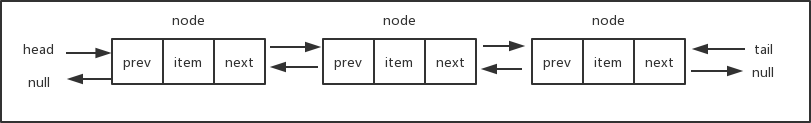
          prev.next = next;

     }

     this.len--;

};

思路二：双链表

[[4]](#endnote-4)

代码：

/\*\*

 \* Initialize your data structure here.

 \*/

var MyLinkedList = function() {

    this.head=null;//头结点

    this.tail=null;//尾结点

};

//辅助类，定义前后指针以及值

var listNode=function(val){

    this.prev=null;

    this.next=null;

    this.val=val;

}

/\*\*

 \* Get the value of the index-th node in the linked list. If the index is invalid, return -1.

 \* @param {number} index

 \* @return {number}

 \*/

MyLinkedList.prototype.get = function(index) {

        let p=this.head;

        while(index--){

            p=p.next;

            if(!p) return -1;//迭代过程中索引节点为空，说明越界

        }

        return p.val;

};

/\*\*

 \* Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.

 \* @param {number} val

 \* @return {void}

 \*/

MyLinkedList.prototype.addAtHead = function(val) {

    //创建新节点

    let cur=new listNode(val);

    //新节点的next指向原来的头节点

    cur.next=this.head;

    //判断原始列表是否为空

    if(this.head){

        this.head.prev=cur;

    }else{

        //如果为空，则将尾指针指向新节点

        this.tail=cur;

    }

    //更新头结点

    this.head=cur;

};

/\*\*

 \* Append a node of value val to the last element of the linked list.

 \* @param {number} val

 \* @return {void}

 \*/

MyLinkedList.prototype.addAtTail = function(val) {

    let cur=new listNode(val)

    if(this.head){

        cur.prev=this.tail

        this.tail.next=cur;

    }else{

        this.head=cur;

    }

this.tail=cur;

};

/\*\*

 \* Add a node of value val before the index-th node in the linked list. If index equals to the length of linked list, the node will be appended to the end of linked list. If index is greater than the length, the node will not be inserted.

 \* @param {number} index

 \* @param {number} val

 \* @return {void}

 \*/

MyLinkedList.prototype.addAtIndex = function(index, val) {

    if(index<=0){

        //加头结点

        this.addAtHead(val)

        return

    }

    let p=this.head;

    while(index--){

        if(!p) return //索引过大，越界

        p=p.next;

    }

    //索引节点为空，说明加的是尾结点

    if(!p){

        this.addAtTail(val);

    }else{

        const cur=new listNode(val);

        //不为头和尾，说明是中间节点

        const preNode=p.prev;

        cur.prev=preNode;

        cur.next=p;

        preNode.next=cur;

        p.prev=cur;

            }

};

/\*\*

 \* Delete the index-th node in the linked list, if the index is valid.

 \* @param {number} index

 \* @return {void}

 \*/

MyLinkedList.prototype.deleteAtIndex = function(index) {

    if(index<0||!this.head) return

    //移除头部元素

    if(index===0){

        //如果链表有多个值

        if(this.head.next){

            this.head.next.prev=null

            this.head=this.head.next

        }else{

            //说明链表只有头节点

            this.head=null;

            this.tail=null;

        }

    }else{

        let p=this.head;

        while(index--){

            p=p.next

            if(!p) return //index大于=链表长度，不合法越界

        }

        if(p.next){

            //index不是尾结点

            p.prev.next=p.next;

            p.next.prev=p.prev

        }else{

            p.prev.next=null

            this.tail=p.prev

        }

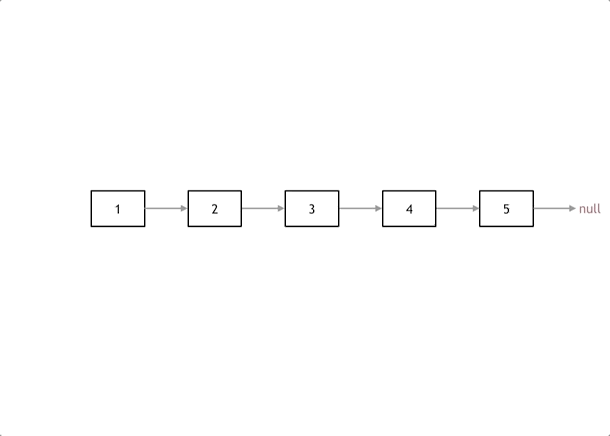
    }

};

注意：双链表的绑定要前后两个都绑定，不然就会报错。

## 反转链表（206题）

思路一：直接将链表翻转。通过双指针，定义前后指针进行翻转（迭代）



代码：

var reverseList = function(head) {

    let cur=head;

    let pre=null;

    while(cur){

        let temp=cur.next;

        cur.next=pre;

        pre=cur;

        cur=temp;

    }

    return pre

};

思路二：递归。

由编译器函数调用执行栈原理可知

最先调用的函数会在递归过程中最后被执行，而最后调用的会最先执行

因此此题，最先返回最后两个节点开始反转操作

依次从后面两两节点开始反转[[5]](#endnote-5)

代码：

var reverseList = function(head) {

    // 如果测试用例只有一个节点 或者 递归到了尾节点，返回当前节点

    if(!head || !head.next) return head;

    // 存储当前节点的下一个节点

    let next = head.next;

    let reverseHead = reverseList(next);

    // 断开 head

    head.next = null;

    // 反转，后一个节点连接当前节点

    next.next = head;

    return reverseHead;

};

## 环形链表（141题）

思路一：通过快慢指针来判断是否有环。慢指针走一步，快指针走两步。如果有环，则总会有遇上的那一刻。

代码：

var hasCycle = function(head) {

    if(!head||!head.next) return false;

    let slow=head;

    let fast=head.next;

    while(fast!=null&&fast.next){

        slow=slow.next;

        fast=fast.next.next;

        if(fast==slow){

            return true

        }

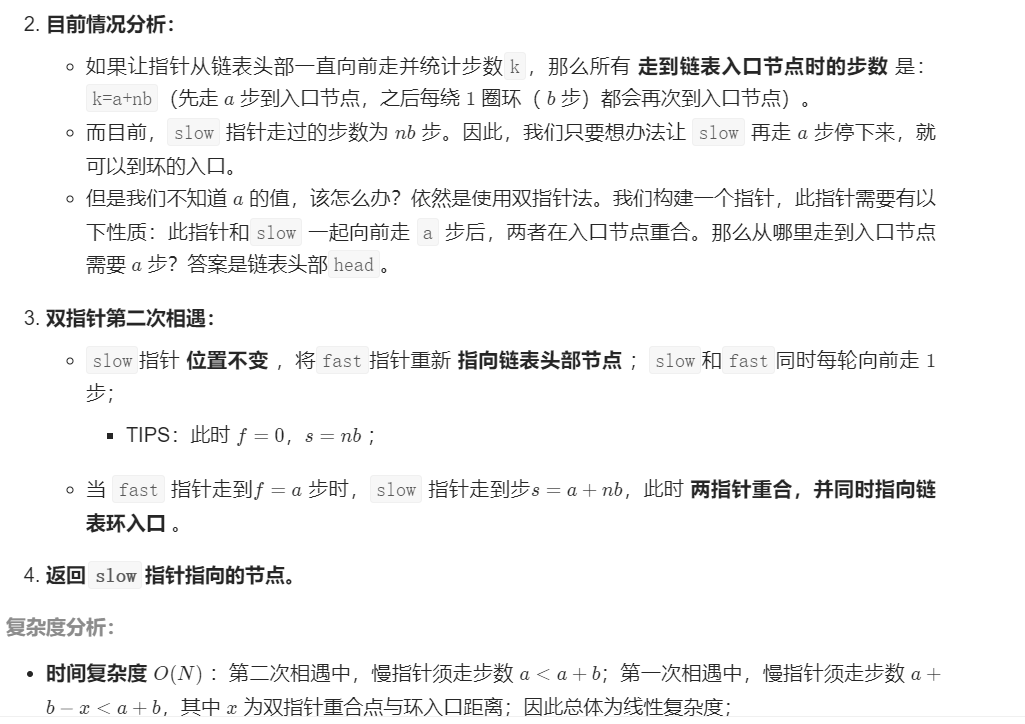
    }

    return false

};

## 环形链表 II（142题）

思路：通过快慢指针来寻找入口点[[6]](#endnote-6)



代码：

var detectCycle = function(head) {

    if(head===null) return null;

    let slow=head,fast=head;

    while(fast!=null){

        slow=slow.next;

        if(fast.next!=null){

            fast=fast.next.next

        }else{

            return null;

        }

        if(slow===fast){

            let ptr=head;

            while(ptr!=slow){

                ptr=ptr.next;

                slow=slow.next;

            }

            return ptr

        }

    }

    return null

};

## 两数相加（2题）

思路：循环两个列表，将列表值相加。需要注意的地方是，十进制会产生进位值，需要将进位值记住，然后加入到和中。还要考虑到l1,l2两个链表的长度不等。此时如果遍历完一条链表，另一个链表已经结束，相加过程中会导致null，所以也要判断一下。

代码：

var addTwoNumbers = function(l1, l2) {

    let flow=0;//是否有进位

    //创建虚拟头结点

    let sum=new ListNode();

    let head=sum;

    while(flow||l1||l2){//在两个链表之中有一个存在的前提下执行下面的逻辑

        let val1=l1?l1.val:0;//取值，需要注意的一个地方是当l1长度<l2时,应该补位为0

        let val2=l2?l2.val:0;

        let r=val1+val2+flow;//求和，十进制中会产生进位

        flow=r>=10?1:0;

        sum.next=new ListNode(r%10);//sum的值

        sum=sum.next;

        if(l1) l1=l1.next;

        if(l2) l2=l2.next;

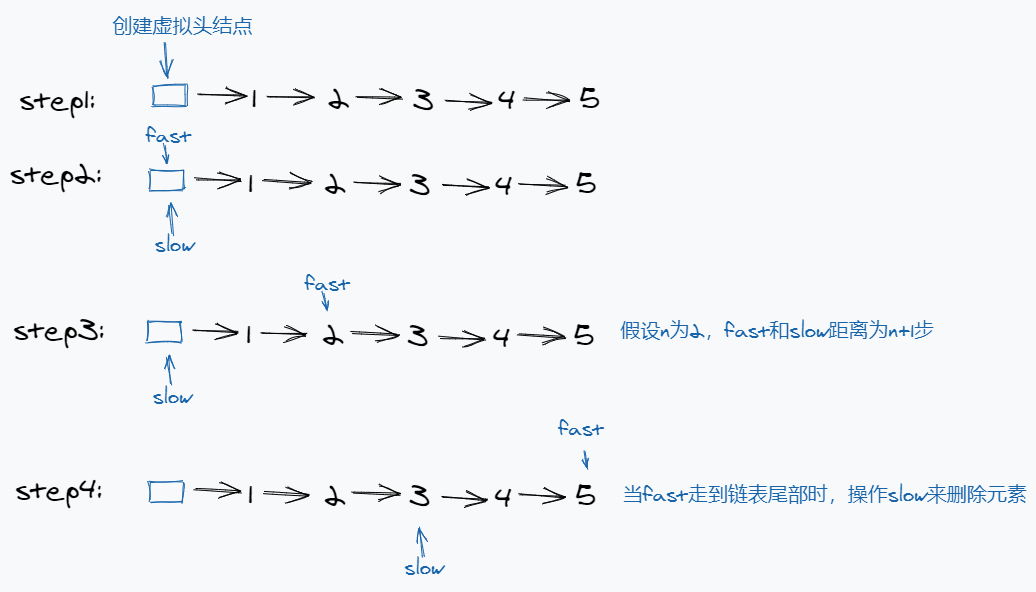
    }

    return head.next

};

## 删除链表的倒数第 N 个结点（19题）

思路：通过双指针的做法，快指针慢指针相差n-1步，然后快慢指针中的位置就是需要删除的元素，在通过慢指针指向进行删除。



代码：

var removeNthFromEnd = function(head, n) {

    if(!head.next&&n===1){return null}

    let dummy=new ListNode(0);

    dummy.next=head;

    let fast=dummy,slow=dummy;

    while(n--){

        fast=fast.next;

    }

    while(fast&&fast.next){

        fast=fast.next;

        slow=slow.next;

    }

    slow.next=slow.next.next;

    return dummy.next;

};

## 合并两个有序链表（21题）

思路:首先创建一个新的链表，和虚拟节点（方便最后返回新的链表）。然后通过next指针指向了l1或l2数值小的那方，最后结束后如果有一方没有合并完，直接将链尾指向为完成的即可，因为已经是排序好了的。

代码：

var mergeTwoLists = function(l1, l2) {

    let dummy=new ListNode(0);

    let pre=dummy;

    while(l1&&l2){

        if(l1.val<=l2.val){

            pre.next=l1;

            l1=l1.next

        }else{

            pre.next=l2;

            l2=l2.next

        }

        pre=pre.next;

    }

    // 合并后 l1 和 l2 最多只有一个还未被合并完，我们直接将链表末尾指向未合并完的链表即可

   pre.next=l1===null?l2:l1;

   return dummy.next;

# 哈希表

# 栈和队列

1. 有效的括号

思路：循环数组，将左括号入栈，右括号与栈中左括号进行匹配，如果不匹配返回false,最后空栈返回。先排除输入为奇数的情况。

[[7]](#endnote-7)算法流程：

双指针第一次相遇： 设两指针 fast，slow 指向链表头部 head，fast 每轮走 22 步，slow 每轮走 11 步；

第一种结果： fast 指针走过链表末端，说明链表无环，直接返回 null；

TIPS: 若有环，两指针一定会相遇。因为每走 11 轮，fast 与 slow 的间距 +1+1，fast 终会追上 slow；

第二种结果： 当fast == slow时， 两指针在环中 第一次相遇 。下面分析此时fast 与 slow走过的 步数关系 ：

设链表共有 a+ba+b 个节点，其中 链表头部到链表入口 有 aa 个节点（不计链表入口节点）， 链表环 有 bb 个节点（这里需要注意，aa 和 bb 是未知数，例如图解上链表 a=4a=4 , b=5b=5）；设两指针分别走了 ff，ss 步，则有：

fast 走的步数是slow步数的 22 倍，即 f = 2sf=2s；（解析： fast 每轮走 22 步）

fast 比 slow多走了 nn 个环的长度，即 f = s + nbf=s+nb；（ 解析： 双指针都走过 aa 步，然后在环内绕圈直到重合，重合时 fast 比 slow 多走 环的长度整数倍 ）；

以上两式相减得：f = 2nbf=2nb，s = nbs=nb，即fast和slow 指针分别走了 2n2n，nn 个 环的周长 （注意： nn 是未知数，不同链表的情况不同）。

目前情况分析：

如果让指针从链表头部一直向前走并统计步数k，那么所有 走到链表入口节点时的步数 是：k=a+nb（先走 aa 步到入口节点，之后每绕 11 圈环（ bb 步）都会再次到入口节点）。

而目前，slow 指针走过的步数为 nbnb 步。因此，我们只要想办法让 slow 再走 aa 步停下来，就可以到环的入口。

但是我们不知道 aa 的值，该怎么办？依然是使用双指针法。我们构建一个指针，此指针需要有以下性质：此指针和slow 一起向前走 a 步后，两者在入口节点重合。那么从哪里走到入口节点需要 aa 步？答案是链表头部head。

双指针第二次相遇：

slow指针 位置不变 ，将fast指针重新 指向链表头部节点 ；slow和fast同时每轮向前走 11 步；

TIPS：此时 f = 0f=0，s = nbs=nb ；

当 fast 指针走到f = af=a 步时，slow 指针走到步s = a+nbs=a+nb，此时 两指针重合，并同时指向链表环入口 。

返回slow指针指向的节点。

复杂度分析：

时间复杂度 O(N)O(N) ：第二次相遇中，慢指针须走步数 a < a + ba<a+b；第一次相遇中，慢指针须走步数 a + b - x < a + ba+b−x<a+b，其中 xx 为双指针重合点与环入口距离；因此总体为线性复杂度；

空间复杂度 O(1)O(1) ：双指针使用常数大小的额外空间。

作者：jyd

链接：https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/linked-list-cycle-ii-kuai-man-zhi-zhen-shuang-zhi-/

来源：力扣（LeetCode）

著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。

1. 作者：xiao\_ben\_zhu

   链接：https://leetcode-cn.com/problems/spiral-matrix/solution/shou-hui-tu-jie-liang-chong-bian-li-de-ce-lue-kan-/

   来源：力扣（LeetCode）

   著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。 [↑](#endnote-ref-1)
2. 作者：xiao\_ben\_zhu

   链接：https://leetcode-cn.com/problems/spiral-matrix/solution/shou-hui-tu-jie-liang-chong-bian-li-de-ce-lue-kan-/ [↑](#endnote-ref-2)
3. https://mp.weixin.qq.com/s?\_\_biz=MzUxNjY5NTYxNA==&mid=2247484132&idx=1&sn=032d3d00bdfb7179941306a2aa50c9f1&chksm=f9a231b5ced5b8a31f0e669ae95bff61626082cd876354c16ca6686d58534ec7926c17e1c3c7&cur\_album\_id=1485825793120387074&scene=189#rd [↑](#endnote-ref-3)
4. https://www.cnblogs.com/jaxu/p/11277732.html [↑](#endnote-ref-4)
5. 作者：Alexer-660

   链接：https://leetcode-cn.com/problems/reverse-linked-list/solution/206-fan-zhuan-lian-biao-by-alexer-660/ [↑](#endnote-ref-5)
6. 作者：jyd

   链接：https://leetcode-cn.com/problems/linked-list-cycle-ii/solution/linked-list-cycle-ii-kuai-man-zhi-zhen-shuang-zhi-/ [↑](#endnote-ref-6)
7. [↑](#endnote-ref-7)