# 3年JAVA技能总结-数据结构篇

1. 常用的List集合：

ArrayList：有序可重复；底层使用数组；查询快增删慢；线程不安全；扩容机制：当前容量 \* 1.5  
LinkedList：有序可重复；底层使用双向链表；查询慢增删快；线程不安全  
Vector：有序可重复；底层使用数组；查询快增删慢；线程安全效率低；扩容机制：容量一倍

1. 常用的Set集合

HashSet：排列无序，不可重复；底层使用hash表实现；存取速度快；内部是HashMap  
LinkedHashSet：采用hash表存储，并用双向链表记录插入顺序；内部是LinkedHashMap  
TreeSet：排列无序，不可重复；底层使用二叉树实现；排序存储；内部是TreeMap的SortedSet

1. 常用的Map

HashMap：key不重复value可重复；底层hash表+链表+红黑树（jdk1.8），hash表+单向链表（jdk1.7）；线程不安全；允许key=null，value=null；1倍扩容  
 如果需要满足线程安全，可以借助Collections工具类的synchroinzedMap方法具有线程安全的能力，或者使用ConcurrentHashMap  
 HashMap的底层数据结构是什么  
 HashMap的put过程  
 HashMap的扩容机制  
 并发环境下HashMap会带来什么致命问题  
 LinkedHashMap：继承HashMap  
 TreeMap：key不重复value可重复；底层二叉树;  
 Hashtable：key不重复value可重复；底层hash表+链表+红黑树（jdk1.8）；线程安全；key和value都不能是null  
 ConcurrentHashMap：  
 锁分段技术：  
 读是否要加锁：  
 迭代器是强一致性的迭代器还是弱一致性的迭代器

CopyOnWrite系列

队列：Queue  
 先入先出FIFO

栈：Stack  
 先入后出FILO

1. JDK1.8 ArrayList源码

public class ArrayList<E> extends AbstractList<E> implements List<E>, RandomAccess, Cloneable{  
 int DEFAULT\_CAPACITY = 10;  
 Object[] EMPTY\_ELEMENTDATA = {};  
 Object[] DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA = {};  
  
 //\*\*\*\*\*\*\*\*\*底层是用数组来存放元素的，扩容方式：3/2倍  
 //底层数组决定了ArrayList是有序的，  
 transient Object[] elementData;  
 int size; //\*\*\*\*\*\*\*\*\*实际大小

//记录该ArrayList被修改的次数，用于fast-fail快速失败机制，该属性在AbstractList中被定义  
 protected transient int modCount = 0;   
  
 //默认构造器，初始化elementData={}  
 public ArrayList() {  
 this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;  
 }  
  
 //指定底层数组的默认大小  
 public ArrayList(int initialCapacity) {  
 if (initialCapacity > 0) {  
 this.elementData = new Object[initialCapacity];  
 } else if (initialCapacity == 0) {  
 this.elementData = EMPTY\_ELEMENTDATA;  
 } else {  
 throw new IllegalArgumentException("Illegal Capacity: "+ initialCapacity);  
 }  
 }  
  
 //ArrayList新增数据的方法 ，元素可以为null  
 //1.由于底层是用数组来存放元素的，所以需要判断是否对数组进行扩容  
 //2.更新modCount++  
 public boolean add(E e) {  
 ensureCapacityInternal(size + 1);  
 elementData[size++] = e;  
 return true;  
 }  
  
 //在指定位置插入元素  
 //1.下标校验 [0,size]就是合法的下标  
 //2.是否考虑需要扩容  
 //3.index后元素全部后移  
 //4.在index位置插入元素  
 //5.更新size  
 public void add(int index, E element) {  
 rangeCheckForAdd(index);  
 ensureCapacityInternal(size + 1);  
 System.arraycopy(elementData, index, elementData, index + 1, size - index);  
 elementData[index] = element;  
 size++;  
 }  
  
 //将底层数组缩放到size大小，因为每次扩容后数组默认被null填充，该方法就是去掉null  
 public void trimToSize() {  
 modCount++;  
 if (size < elementData.length) {  
 elementData = (size == 0) ? EMPTY\_ELEMENTDATA : Arrays.copyOf(elementData, size);  
 }  
 }  
  
 //判断该list是否包含指定元素  
 public boolean contains(Object o) {  
 return indexOf(o) >= 0;  
 }  
  
 //返回该list指定元素的下标，如果有重复的就返回从左到右第一个  
 public int indexOf(Object o) {  
 if (o == null) {  
 for (int i = 0; i < size; i++)  
 if (elementData[i]==null)  
 return i;  
 } else {  
 for (int i = 0; i < size; i++)  
 if (o.equals(elementData[i]))  
 return i;  
 }  
 return -1;  
 }  
  
 public E get(int index) {  
 rangeCheck(index);  
 return elementData(index);  
 }  
  
 //移除指定位置的元素  
 public E remove(int index) {  
 rangeCheck(index);  
 modCount++;  
 E oldValue = elementData(index);  
 int numMoved = size - index - 1;  
 if (numMoved > 0) //数据前移  
 System.arraycopy(elementData, index+1, elementData, index, numMoved);  
 elementData[--size] = null;

return oldValue;  
 }  
  
 private void ensureCapacityInternal(int minCapacity) {  
 ensureExplicitCapacity(calculateCapacity(elementData, minCapacity));  
 }  
  
 private static int calculateCapacity(Object[] elementData, int minCapacity) {  
 if (elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {  
 return Math.max(DEFAULT\_CAPACITY, minCapacity);  
 }  
 return minCapacity;  
 }  
  
 private void ensureExplicitCapacity(int minCapacity) {  
 modCount++;  
 if (minCapacity - elementData.length > 0)  
 grow(minCapacity);  
 }  
  
 private void grow(int minCapacity) {  
 int oldCapacity = elementData.length; //当前数组的容量  
 int newCapacity = oldCapacity + (oldCapacity >> 1); //新容量= 3/2倍的老容量  
 if (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;  
 if (newCapacity - MAX\_ARRAY\_SIZE > 0)  
 newCapacity = hugeCapacity(minCapacity);  
 // minCapacity is usually close to size, so this is a win:  
 elementData = Arrays.copyOf(elementData, newCapacity); //将老数组中的元素全部复制到新数组中  
 }  
  
 private static int hugeCapacity(int minCapacity) {  
 if (minCapacity < 0) throw new OutOfMemoryError();  
 return (minCapacity > MAX\_ARRAY\_SIZE) ? Integer.MAX\_VALUE : MAX\_ARRAY\_SIZE;  
 }  
}

1. JDK1.8 LinkedList源码

public class LinkedList<E> extends AbstractSequentialList<E> implements List<E>, Deque<E>, Cloneable{  
  
 transient int size = 0;  
  
 //底层维护了一个双向链表，快速删除和插入，查询比较慢  
 transient Node<E> first; //双向链表的头结点  
 transient Node<E> last; //双向链表的尾节点  
  
 protected transient int modCount = 0; //fast-fail机制  
  
 private static class Node<E> { //双向链表  
 E item;  
 Node<E> next;  
 Node<E> prev;  
 }  
  
 //核心方法：新增元素，链表尾部新增节点  
 public boolean add(E e) {  
 linkLast(e);  
 return true;  
 }  
  
 void linkLast(E e) {  
 final Node<E> l = last;  
 final Node<E> newNode = new Node<>(l, e, null); //新增节点  
 last = newNode; //更新尾节点  
 if (l == null)  
 first = newNode; //初始化头结点  
 else  
 l.next = newNode;  
 size++;  
 modCount++;  
 }  
  
 //获取链表中指定位置的元素  
 public E get(int index) {  
 checkElementIndex(index);  
 return node(index).item;  
 }  
  
 //获取指定索引位置的节点  
 Node<E> node(int index) {  
 if (index < (size >> 1)) {  
 Node<E> x = first;  
 for (int i = 0; i < index; i++)  
 x = x.next;  
 return x;  
 } else {  
 Node<E> x = last;  
 for (int i = size - 1; i > index; i--)  
 x = x.prev;  
 return x;  
 }  
 }  
  
 public void add(int index, E element) {  
 checkPositionIndex(index);  
 if (index == size)  
 linkLast(element); //直接在链表尾部新增节点  
 else  
 linkBefore(element, node(index)); //链表中间插入节点  
 }  
  
 void linkBefore(E e, Node<E> succ) {  
 final Node<E> pred = succ.prev;  
 final Node<E> newNode = new Node<>(pred, e, succ);  
 succ.prev = newNode;  
 if (pred == null)  
 first = newNode;  
 else  
 pred.next = newNode;  
 size++;  
 modCount++;  
 }  
}

1. JDK1.8 Vector源码

public class Vector<E>{  
 protected Object[] elementData; //底层使用数组来存储元素  
 protected int elementCount; //等同于size  
 protected int capacityIncrement; //数组扩增的增量大小  
  
 public Vector() {  
 this(10); //默认初始化底层数组大小为10，增量为0  
 }  
  
 public Vector(int initialCapacity) {  
 this(initialCapacity, 0);  
 }  
  
 public Vector(int initialCapacity, int capacityIncrement) {  
 super();  
 this.elementData = new Object[initialCapacity]; //初始化底层数组  
 this.capacityIncrement = capacityIncrement; //初始化底层数组的扩容增量  
 }  
  
 public synchronized boolean add(E e) { //同步，效率低下  
 modCount++;  
 ensureCapacityHelper(elementCount + 1); //考虑是否扩容数组  
 elementData[elementCount++] = e;  
 return true;  
 }  
  
 private void ensureCapacityHelper(int minCapacity) {  
 if (minCapacity - elementData.length > 0) //考虑是否需要扩容  
 grow(minCapacity);  
 }  
  
 private void grow(int minCapacity) { //minCapacity是数组的最小容量  
 int oldCapacity = elementData.length;  
 //默认的扩容规则：当前容量 + 增量  
 int newCapacity = oldCapacity + ((capacityIncrement > 0) ? capacityIncrement : oldCapacity);  
 if (newCapacity - minCapacity < 0)  
 newCapacity = minCapacity;  
 if (newCapacity - MAX\_ARRAY\_SIZE > 0)  
 newCapacity = hugeCapacity(minCapacity);  
 elementData = Arrays.copyOf(elementData, newCapacity);  
 }  
}

1. 单向链表

定义

private static class ListNode<T> {

T val;

ListNode<T> next;

public ListNode(T val) {

this.val = val;

}

}

头插法创建单向链表

//头插法创建单向链表：每次生成的新节点设置为头节点

private static ListNode<Integer> headLink() {

ListNode<Integer> header = null;

for (int i = 0; i < 10; i++) {

ListNode<Integer> newNode = new ListNode<>(random.nextInt(10) + 1);

if (header == null) {

header = newNode;

} else {

//将新节点放在header的前面

newNode.next = header;

//更新新节点就是头节点

header = newNode;

}

}

return header;

}

//头插法  
 private static ListNode headInsert() {

ListNode head;  
 if (head == null) {  
 head = new ListNode(); //特殊，头结点不存储任何数据  
 }  
 for (int i = 0; i < 10; i++) {  
 ListNode newNode = new ListNode(random());  
 newNode.next = head.next;  
 head.next = newNode;  
 }

return head;  
}

尾插法创建单向链表

//尾插法创建单向链表：每次生成的新节点设置为尾节点  
private static ListNode<Integer> tailLink() {  
 ListNode<Integer> header = null, tailer = null;  
 for (int i = 0; i < 10; i++) {  
 ListNode<Integer> newNode = new ListNode<>(random.nextInt(10) + 1);  
 if (header == null) {  
 header = tailer = newNode;  
 } else {  
 //尾节点后面插入新节点  
 tailer.next = newNode;  
 //更新新节点就是尾节点  
 tailer = newNode;  
 }  
 }  
 return header;  
}

//尾插法  
private static ListNode tailInsert() {  
 ListNode tail = null;  
 if (head == null) {  
 head = new ListNode(); //特殊，头结点不存储任何数据  
 tail = head;  
 }  
 for (int i = 0; i < 10; i++) {  
 ListNode newNode = new ListNode(random());  
 tail.next = newNode;  
 tail = newNode;  
 }

return head;  
}

正序输出单向链表

private static void positivePrint(ListNode head) {  
 if (head != null) {  
 System.out.println();  
 ListNode newHead = head;  
 while (newHead != null) {  
 System.out.print(newHead.val + "->");  
 newHead = newHead.next;  
 }  
 }  
}

单向链表逆序输出

//将单向链表逆序，改变原链表，参考https://blog.csdn.net/alpgao/article/details/86509265  
//借助头结点  
private static ListNode reverse() {  
 if (head == null || head.next == null) {  
 return head;  
 }  
 ListNode p = head.next, q = p.next, next = null;  
 p.next = null;  
 while (q != null) {  
 next = q.next;  
 q.next = p;  
 p = q;  
 q = next;  
 }  
 head.next = p;  
 return head;  
}

// 在不使用额外存储节点的情况下使一个单链表的所有节点逆序  
// 循环式 https://blog.csdn.net/xiao\_ma\_CSDN/article/details/80550092  
private static ListNode reverse1(ListNode head) {  
 if (head == null || head.next == null) {  
 return head;  
 }  
 ListNode curHead = head, prev = null, next = null;  
 while (curHead != null) {  
 //记录下当前节点的下一个节点  
 next = curHead.next;  
 //逆序  
 curHead.next = prev;  
 //后移  
 prev = curHead;  
 //后移  
 curHead = next;  
 }  
 return prev;  
}

//在不使用额外存储节点的情况下使一个单链表的所有节点逆序  
//递归方式https://blog.csdn.net/xiao\_ma\_CSDN/article/details/80550092  
private static ListNode reverse2(ListNode head) {  
 if (head == null || head.next == null) {  
 return head;  
 }  
 ListNode newHead = reverse2(head.next);  
 //另当前节点的下一个节点指向当前节点，形成闭合回路  
 head.next.next = head;  
 //断开闭合回路，实现逆序  
 head.next = null;  
 return newHead;  
}

//利用栈的先入后出特性实现倒序打印，并没有改变原链表  
private static void reverse3(ListNode head) {  
 ListNode newNode = head;  
 Stack<ListNode> stack = new Stack<>(); //利用栈的FILO特性  
 while (newNode != null) {  
 stack.push(newNode); //入栈  
 newNode = newNode.next;  
 }  
 System.out.println();  
 while (stack.size() > 0) {  
 newNode = stack.pop(); //出栈  
 System.out.print(newNode.val + "->");  
 }  
}

//链表入栈  
private static void reverse4(ListNode head) {  
 if (head == null || head.next == null) {  
 return;  
 }  
 ListNode curNode = head, next = null;  
 Stack<ListNode> stack = new Stack<>();  
 while (curNode != null) {  
 //将当前节点入栈  
 stack.push(curNode);  
 //找到next节点  
 next = curNode.next;  
 //断开下一个节点  
 curNode.next = null;  
 //后移  
 curNode = next;  
 }  
  
 head = stack.pop();  
 curNode = head;  
 while (!stack.isEmpty()) {  
 curNode.next = stack.pop();  
 curNode = curNode.next;  
 }  
}

1. 双向链表

定义

private static class DoubleListNode<T> {  
 T val;  
 DoubleListNode next;  
 DoubleListNode prev;  
 public DoubleListNode(T val) {  
 this.val = val;  
 }  
 }

头插法创建双向链表

private static void headInsert() {  
 DoubleListNode next = null, head = null;  
 for (int i = 0; i < 10; i++) {  
 DoubleListNode<Integer> newNode = new DoubleListNode(i);  
 if (head == null) {  
 head = newNode;  
 } else {  
 next = head.next;  
 if (next != null) {  
 next.prev = newNode;  
 }  
 newNode.next = next;  
 head.next = newNode;  
 newNode.prev = head;  
 }  
 }

return head;  
}

尾插法创建双向链表

private static void tailInsert() {  
 DoubleListNode tail = null, head = null;  
 for (int i = 0; i < 10; i++) {  
 DoubleListNode<Integer> newNode = new DoubleListNode(i);  
 if (head == null) {  
 head = tail = newNode;  
 } else {  
 tail.next = newNode;  
 newNode.prev = tail;  
 tail = newNode;  
 }  
 }

return head;  
}

LinkedHashMap

TreeMap

ConcurrentHashMap

HashSet

TreeSet

LinkedHashSet

HashMap

//计算大于目标容量最小的2次幂，最大2^30

static final int tableSizeFor(int cap) {

int n = cap - 1;

n |= n >>> 1;

n |= n >>> 2;

n |= n >>> 4;

n |= n >>> 8;

n |= n >>> 16;

return (n < 0) ? 1 : (n >= MAXIMUM\_CAPACITY) ? MAXIMUM\_CAPACITY : n + 1;

}

//计算key的hash值，用于散列分布到数组索引上

//计算方式：高16位与低16位做异或运算

//目的：int index = (n - 1) & hash(key);较少碰撞，散列均匀

//如果直接使用key的hashcode()作为hash很容易发生碰撞。比如，在n - 1为15(0x1111)时，散列值真正生效的只是低4位。当新增的键的hashcode()是2，18，34这样恰好以16的倍数为差的等差数列，就产生了大量碰撞。

static int hash(Object k){

int h;

return k == null ? 0 : ((h = k.hashCode()) ^ (h >>> 16));

}

1. 树

二叉树

定义：每个结点至多只有二棵子树(不存在度大于2的结点)  
性质： 二叉树的第i层至多有2^(i-1)个结点；

深度为k的二叉树至多有2^k-1个结点

public static class TreeNode<T> {

private T val;

private TreeNode left;

private TreeNode right;

}

满二叉树

定义：除最后一层无任何子节点外，每一层上的所有结点都有两个子结点。也可以这样理解，除叶子结点外的所有结点均有两个子结点。节点数达到最大值，所有叶子结点必须在同一层上。  
性质： 一颗树深度为h，最大层数为k，深度与最大层数相同，k=h;  
 叶子数为2h;  
 第k层的结点数是2^(k-1);  
 总结点数是2^k-1，且总节点数一定是奇数。

完全二叉树

定义：若设二叉树的深度为h，除第h层外，其它各层 (1～(h-1)层) 的结点数都达到最大个数，第h层所有的结点都连续集中在最左边。

二叉查找树(二叉搜索树、二叉排序树)  
 若左子树不空，则左子树上所有结点的值均小于它的根结点的值；  
 若右子树不空，则右子树上所有结点的值均大于或等于它的根结点的值；  
 左、右子树也分别为二叉排序树；  
 没有键值相等的节点。  
性质： 对二叉查找树进行中序遍历，即可得到有序的数列。  
 二叉查找树的时间复杂度：它和二分查找一样，插入和查找的时间复杂度均为O(logn)，但是在最坏的情况下仍然会有O(n)的时间复杂度。原因在于插入和删除元素的时候，树没有保持平衡。我们追求的是在最坏的情况下仍然有较好的时间复杂度，这就是平衡查找树设计的初衷。 二叉查找树的高度决定了二叉查找树的查找效率

平衡二叉树AVL

它是一棵空树或它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树

红黑树BRT：一种含有红黑结点并能自平衡的二叉查找树  
性质： 每个节点要么是黑色，要么是红色

根节点是黑色

每个叶子节点（NIL）是黑色

每个红色结点的两个子结点一定都是黑色

任意一结点到每个叶子结点的路径都包含数量相同的黑结点

从性质5又可以推出：

如果一个结点A存在黑子结点，那么该结点A肯定有两个子结点

红黑树自平衡的三种操作：左旋、右旋和变色

左旋：以某个结点作为支点(旋转结点)，其右子结点变为旋转结点的父结点，右子结点的左子结点变为旋转结点的右子结点，左子结点保持不变。

右旋：以某个结点作为支点(旋转结点)，其左子结点变为旋转结点的父结点，左子结点的右子结点变为旋转结点的左子结点，右子结点保持不变。

变色：结点的颜色由红变黑或由黑变红。

二叉树的遍历方式

前序（根、左子树、右子树）、中序（左子树、根、右子树）、后序（左子树、右子树、根）、层次（从上到下，从左到右）

2-3树

1. Tree

B+Tree

哈夫曼树

二叉树遍历（

前序（根左右）、中序（左根右）、后序（左右根）、层次、深度优先、广度优先遍历）

ArrayDeque{

//ArrayDeque是一个用循环数组实现的双端队列

//不允许存null（抛NullPointerException），计算容量算法，扩容算法

//初始化时计算容量：大于默认容量或用户指定容量的最小的2的幂次方

//新增元素时扩容：双倍

transient Object[] elements;

transient int head; //头部节点序号

transient int tail; //尾部节点序号，（指向最后一点节点的后一个位置）

private static final int MIN\_INITIAL\_CAPACITY = 8; //允许最小的elements的容量，限制用户自定义

//当head==tail时扩容

public ArrayDeque() {

elements = new Object[16]; //16是比8大的最小的2的幂次方

}

public ArrayDeque(int numElements){

allocateElements(numElements); //计算elements容量并初始化elements

}

public void addLast(E e) {

if(e == null) throw new NullPointerException();

elements[tail] = e;

if( (tail = (tail + 1) & (elements.size - 1)) == head) {

doubleCapacity(); //双倍扩容

}

}

//转成补码计算：正数原码反码补码相同，负数反码为符号位置1，其余位按位取反，补码为反码+1

public void addFirst(E e){

if (e == null) throw new NullPointerException();

elements[head = (head - 1) & (elements.length - 1)] = e; //注意：head是-1

if (head == tail)

doubleCapacity();

}

因为elements.length是2的幂次方，所以减一后就变成了掩码，tail如果记录的是最后一个位置，即 elements.length - 1，tail + 1 则等于elements.length，与 elements.length - 1 做与操作后，就变成了0，嗯，没错，这样就变成了一个循环数组，如果tail与head相等，则表示没有剩余空间可以存放更多元素了，则调用doubleCapacity进行扩容：

private void doubleCapacity() {

assert head == tail;

int p = head;

int n = elements.length();

int r = n - p;

int newCapacity = n << 1; //新的容量，有符号位左移，低位置0

if (newCapacity < 0) {

throw new IllegalStateException(“Sorry,deque is too big!”);

}

Object[] newElements = new Object[newCapacity];

//将老数组的[head, elements.size()]放在新数组的[0, element.size() - head]上

System.copyarray(elements, p, newElements, 0, r);

//将老数组的[0, head]放在新数组的[element.size() - head, elements.size()]上

System.copyarray(elements, 0, newElements, r, p);

//置换原因：addFirst时head-1，addLast时tail+1，位置循环

elements = newElements;

head = 0;

tail = n;

}

public int size() {

return (tail - head) & (elements.length() - 1);

}

public boolean isEmpty() {

return head == tail;

}

//计算elements的初始容量:大于numElements的最小的2的幂次方

//调用：初始化ArrayDeque实例时、反序列化时

//原理：

\* 只需要关注二进制中第一个高位为1的位即可，其余为可以忽略

\* 1.假设在第n位（...... n n-1 n-2 ...... 2 1）

\* 2.无符号右移1位再按位或运算，则n n-1都是1 -->有2位为1

\* 3.无符号右移2位再按位或运算，则n ...... n-3都是1 -->有4位为1

\* 4.无符号右移4位再按位或运算，则n ...... n-7都是1 -->有8位为1

\* 5.无符号右移8位再按位或运算，则n ...... n-15都是1 -->有16位为1

\* 6.无符号右移16位再按位或运算，则n ...... n-31都是1 -->有32位为1

\* 7.java int类型4字节，32位，正数符号位为0，且无符号右移时最多31位为1，即最大生成Interger.MAX\_VALUE=0x7FFFFFFF

@Test

public void testArrayDeque() {

System.out.println(Integer.MAX\_VALUE);

System.out.println(Integer.MAX\_VALUE + 1);

allocateElements(Integer.MAX\_VALUE);

}

//计算时转成补码

private void allocateElements(int numElements) {

int initialCapacity = numElements;

if (initialCapacity > MIN\_INITIAL\_CAPACITY) { //限制最小容量为8

initialCapacity |= initialCapacity >>> 1; //无符号右移1位再按位或

initialCapacity |= initialCapacity >>> 2; //无符号右移2位再按位或

initialCapacity |= initialCapacity >>> 4; //无符号右移4位再按位或

initialCapacity |= initialCapacity >>> 8; //无符号右移8位再按位或

initialCapacity |= initialCapacity >>> 16; //无符号右移16位再按位或

initialCapacity ++;

if(initialCapacity < 0) {

initialCapacity >>>= 1;

}

}

elements = new Object[initialCapacity]; //生成null数组

}

}

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | ... | i | ... | head | ... | tail | ... |