时长：25h

## 欢迎学习《玩转数据结构》

### 欢迎学习《玩转数据结构》

### 学习数据结构(和算法)到底有没有用？

### 关于课程学习的更多注意事项

### 课程编程环境搭建

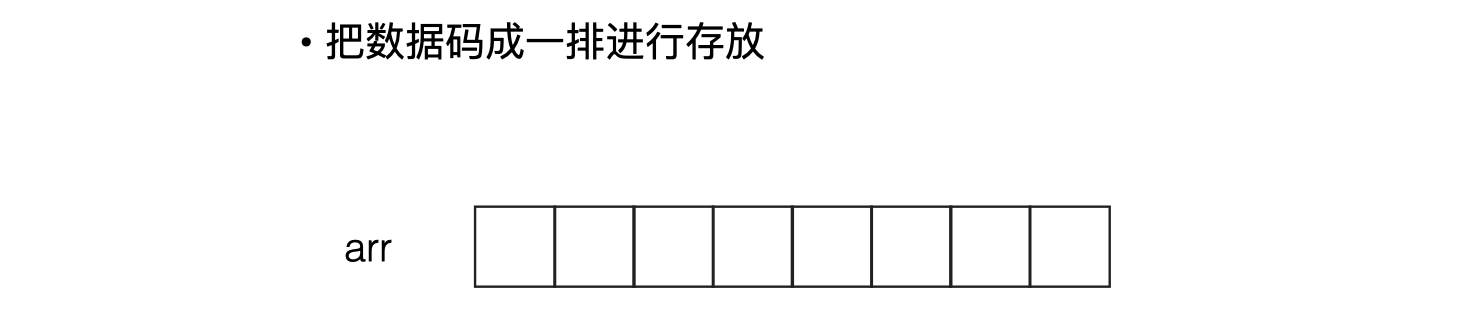
### 关于课程的其他语言支持(Python,C++,JS,Go)

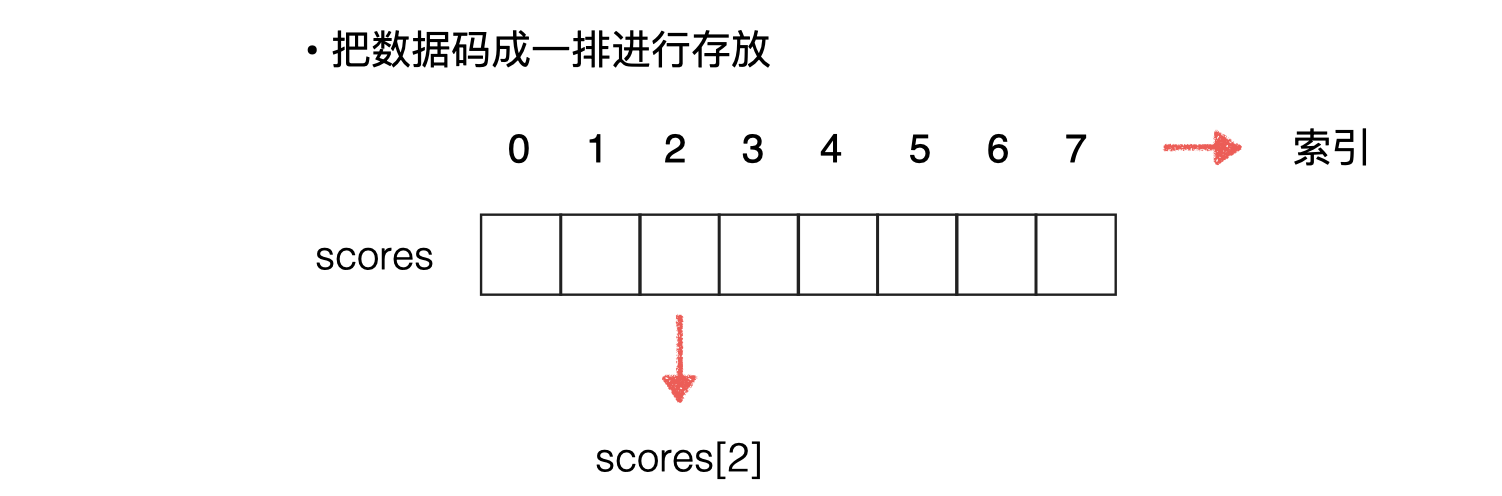
### 在学习数据结构的具体知识前，你可能想读一读这两篇文章

## 不要小瞧数组

### 使用 Java 中的数组

》》先复习下数组的基础知识





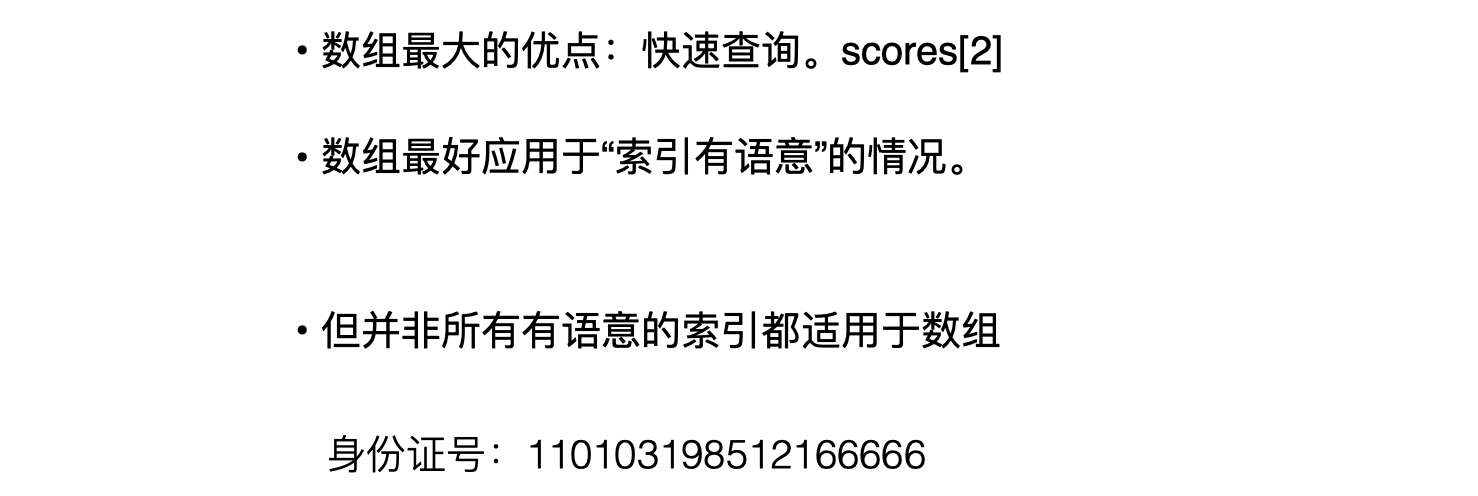
### 二次封装属于我们自己的数组

》》这里对数组进行封装，用我们的意义来进行

A screenshot of a cell phone

Description automatically generated

这里对数据的索引进行一下解释，数组既可以有语义也可以不用有语义

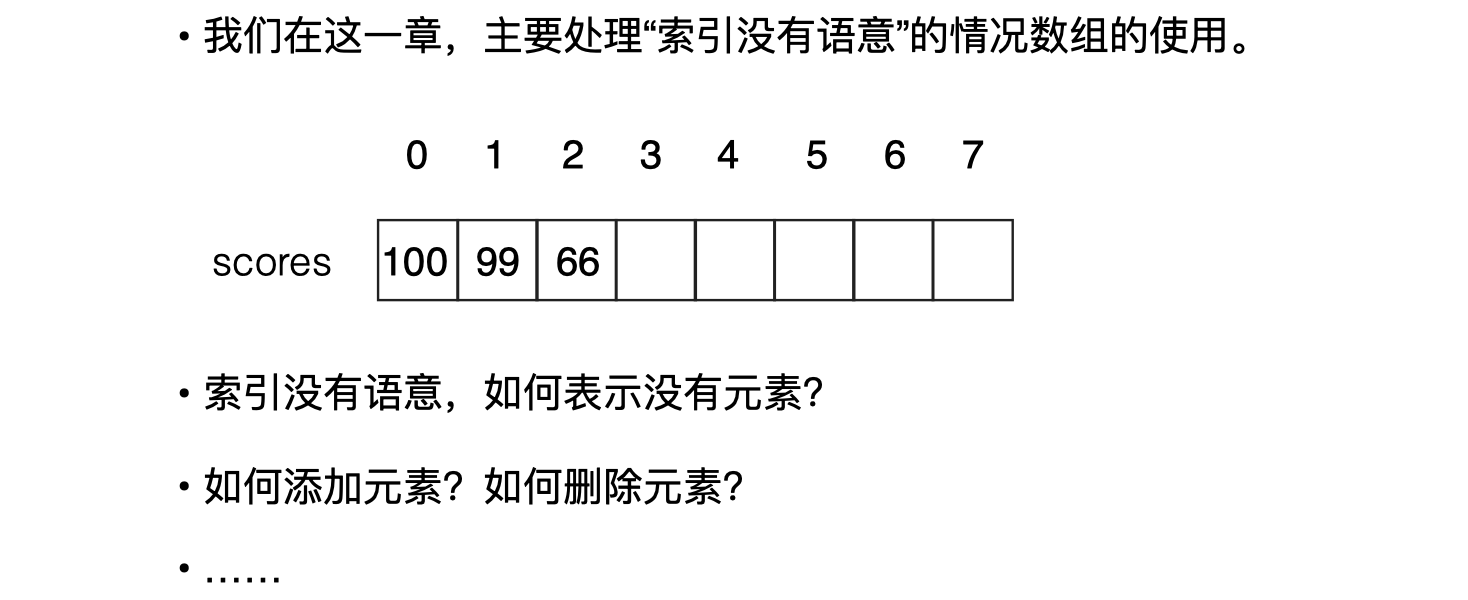


在本章中，我们使用的数组都是没有语意的

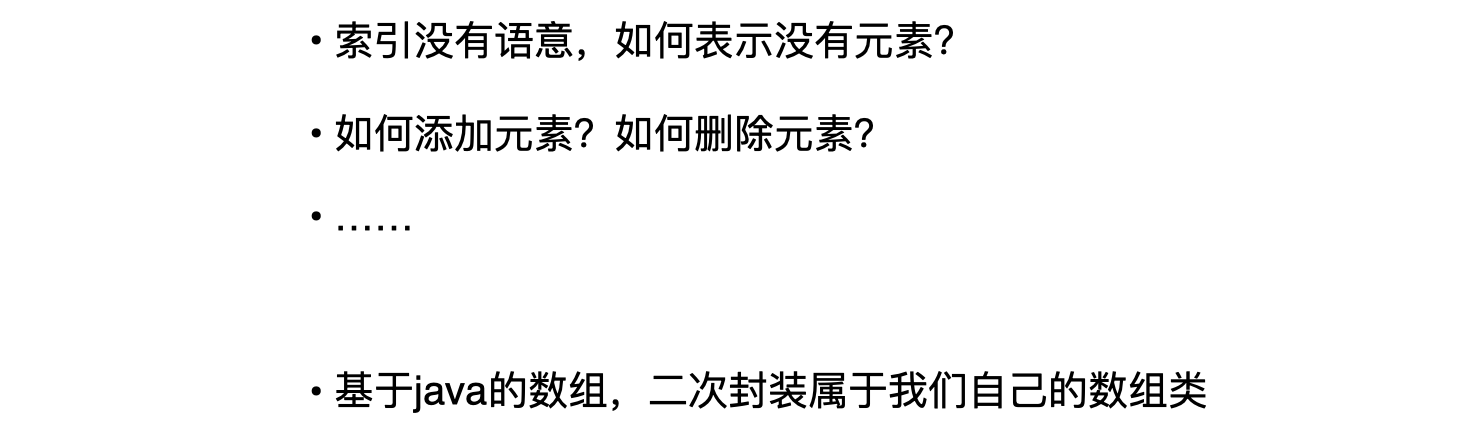
A screenshot of a cell phone

Description automatically generated

那既然没有意义，下面的问题如何处理了?



答案就是我们**封装自己的数组**



具体的我们的数组会有如下的方法

A screenshot of a cell phone

Description automatically generated

》》 下面具体编写代码

|  |
| --- |
| public class Array {  private int[] data; // 数据  private int size; // 多少个元素    private static final int DEFAULT\_CAPACITY = 10;    public Array(int capacity) {  this.data = new int[capacity];  this.size = 0;  }  public Array() {  this(DEFAULT\_CAPACITY);  }  // 获取数组元素个数  public int size(){  return this.size;  }  // 获取数组容量  public int getCapacity() {  return this.data.length;  }  // 数组是否为空  public boolean isEmpty() {  return this.size == 0;  }  } |

### 向数组中添加元素

这节课像我们的数组中添加元素，分为**向末尾添加元素**、向**任意位置添加元素**

**void addFirst(int item); // 数组首部添加元素**

**void adddLast(int item); // 数组尾部添加元素**

**void add(int index, int item); // 任意位置添加元素**

》》首先是在**数组末尾添加元素**

A screenshot of a cell phone

Description automatically generated

\*\* 代码的编写

|  |
| --- |
| void addLast(int item) {  data[size] = item;  size++;  } |

》》然后是在**任意位置添加元素**

中间的截图如下

A screenshot of a cell phone

Description automatically generated

\*\* 代码的编写

// 是否需要先检查容量

|  |
| --- |
| void add(int index, int item){  for(int i = size; i >= index ; i --) {  data[i] = data[i-1];  }  data[index] = item;  } |

### 数组中查询元素和修改元素

编写代码

在数组中查询元素和修改元素，这个通过序号就可以解决了

**int get(int index); // 获取元素**

**voie set(int index, int item); // 更新元素**

代码如下：

|  |
| --- |
| int get(int index){  return data[index];  }  int set(int index, int item) {  data[index] = item;  } |

### 包含、搜索和删除

具体要实现的代码是：

**boolean contains(int item); // 是否包含元素**

**int find(int item;); // 查找元素所在的索引位置**

**void remove(int index); // 移除 index 位置的元素**

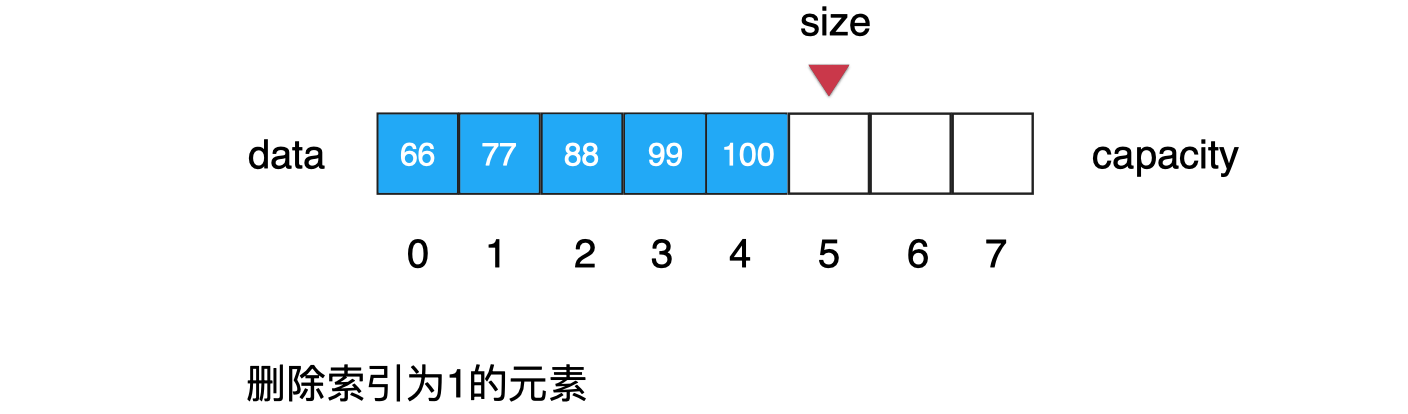
**void removeFirst();**

**void removeLast();**

》》从数组中删除元素

删除指定位置元素

如下图就是，删除 77 元素，删除完以后，需要将后面的元素一个一个的往前移，最后别忘了 size--。



具体代码如下

|  |
| --- |
| public boolean contais(int item){  for(int i = 0;i < size;i++) {  if(data[i] == item) {  return true;  }  }  return false;  }  public int find (int item){  for(int i = 0;i < size;i++) {  if(data[i] == item) {  return i;  }  }  return 0;  }  public void remove(int index) {  for(int i = index; i < size; i++) {  data[i] = data[i + 1];  }  data[size] = 0;  } |

### 使用泛型

上面的数组支持 int 类型，这里我们进行扩展，让数组可以支持的类型更多

》》如下图所示

A screenshot of a cell phone

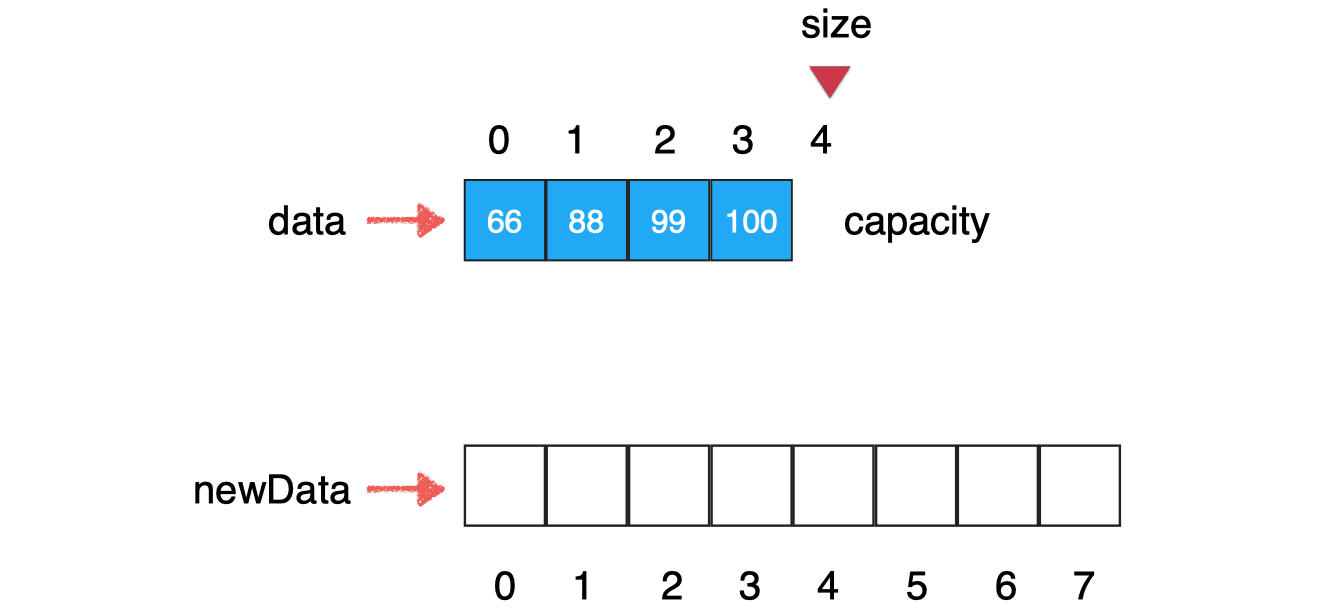
Description automatically generated

》》修改上面的代码，让我们的数组支持泛型

|  |
| --- |
|  |

### 动态数组

》》看下图

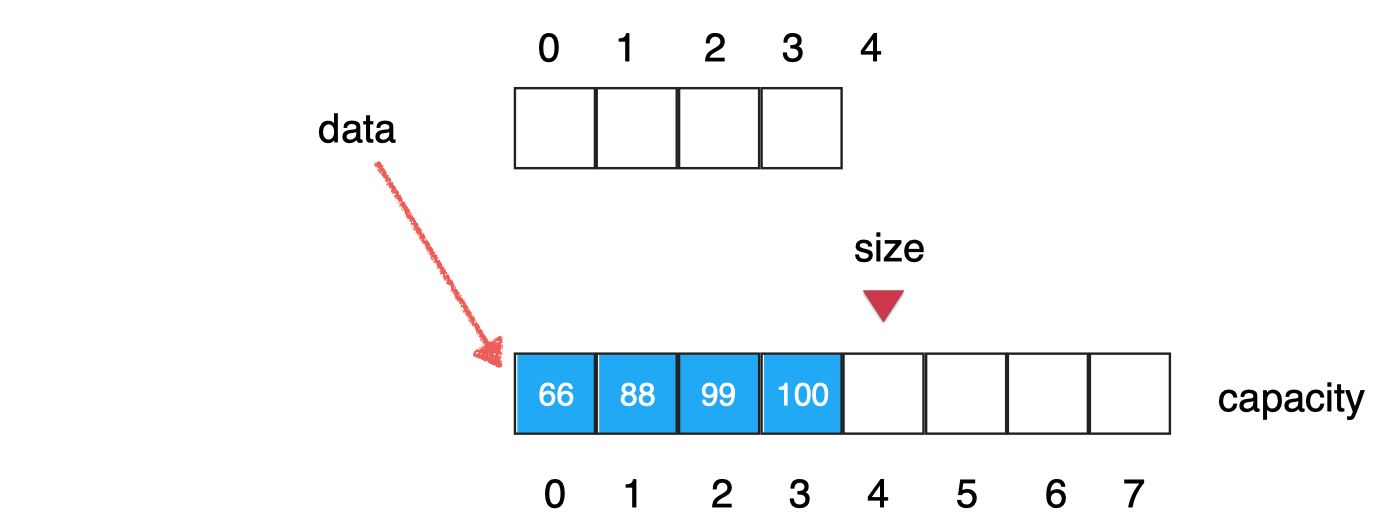


data 到 newdata 的就相当于数组扩容操作

A screenshot of a cell phone

Description automatically generated

最后将 data指向新的数组，就扩容了数组，就相当于动态数组



》》下面进行动态数组的代码的编写

|  |
| --- |
|  |

### 简单的复杂度分析

》》先介绍下时间复杂度的概念

A screenshot of a cell phone

Description automatically generated

》》为什么实用大 O 的方法来了？

A screenshot of a cell phone

Description automatically generated

》》下面看几个具体的例子

A screenshot of a cell phone

Description automatically generated

》》下面分析下动态数组的时间复杂度

A screenshot of a cell phone

Description automatically generated

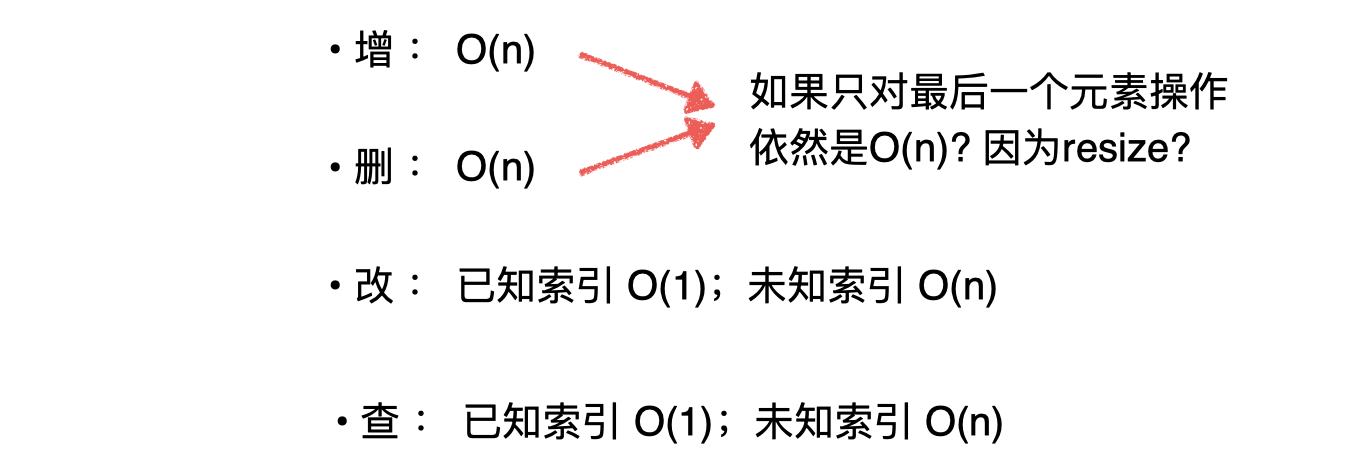
A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated





### 均摊复杂度和防止复杂度的震荡

》》这里我们进行resize 的复杂度分析

A picture containing object

Description automatically generated

A screenshot of a cell phone

Description automatically generated

\*\* 举例说明：

A screenshot of a cell phone

Description automatically generated

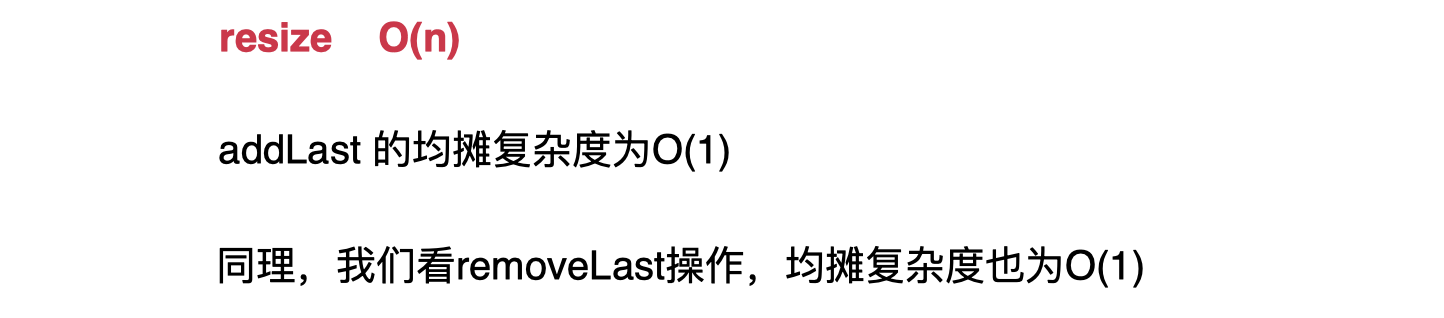
A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

》》均摊复杂度(amortized time complexity)



》》复杂度震荡

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

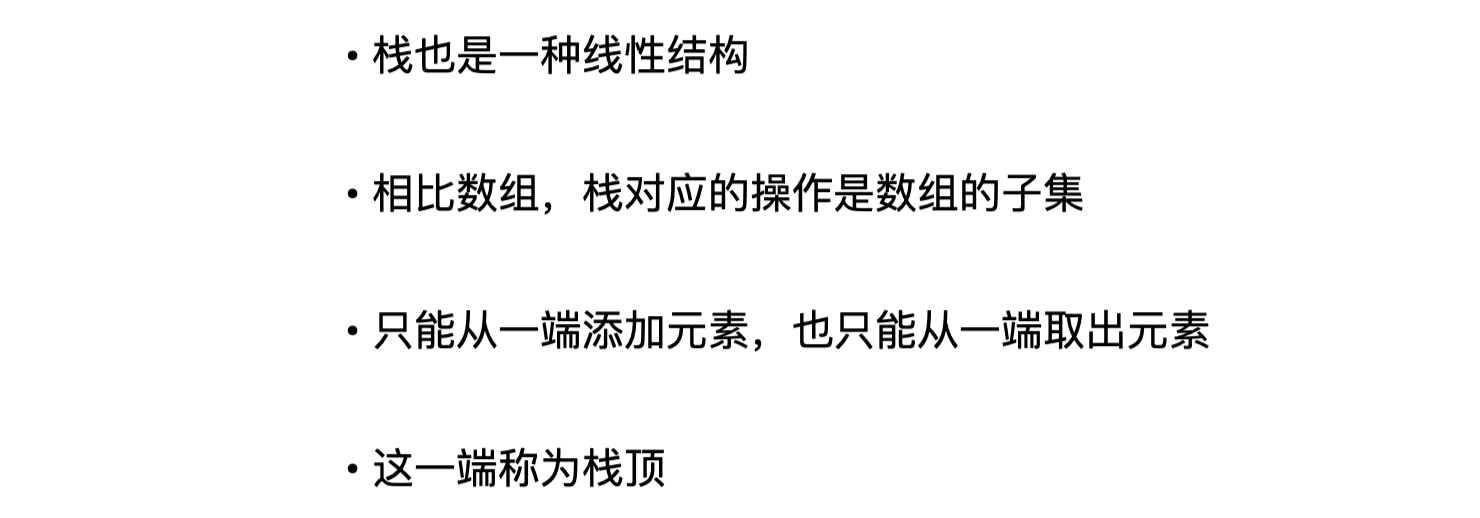
Description automatically generated

》》这里改造代码防止复杂度震荡

## 栈和队列

### 栈和栈的应用：撤销操作和系统栈

》》栈的介绍

》》栈的特性

A screenshot of a cell phone

Description automatically generated

》》栈的应用

\*\* 如 word 里面的撤销操作

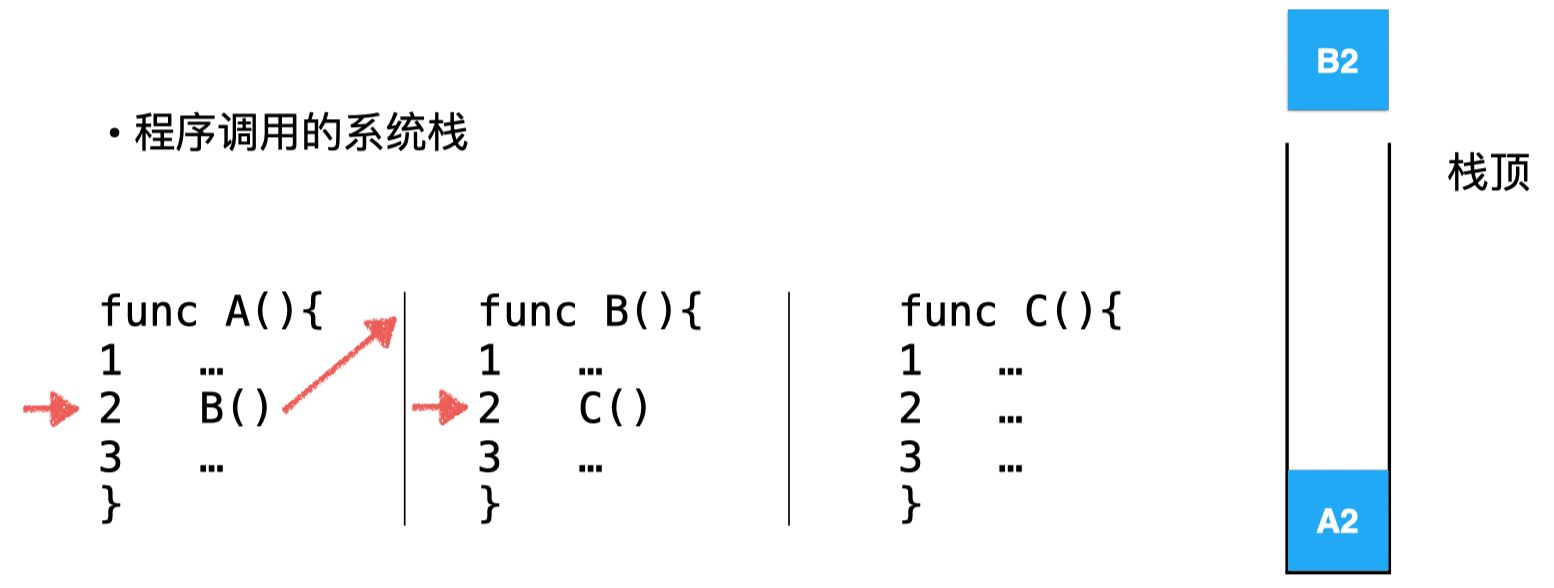
A screenshot of a cell phone

Description automatically generated

\*\* 程序调用的系统栈

这里只是一个截图，多张 ppt 演示了程序如何入栈、出栈的一个效果

如下图：程序在执行 C 完以后，B2 就会进行出栈，继续执行 B 的方法。



### 栈的基本实现

》》栈的基本方法如下

push：压栈

pop： 为弹栈，删除栈顶的值

peek： 返回栈顶的元素，不删除栈顶的值

getSize：获取栈的大小

isEmpty：判断栈是否为空

A screenshot of a cell phone

Description automatically generated

》》使用接口的方式来实现

如下使用数组来实现栈

A screenshot of a cell phone

Description automatically generated

》》具体的代码如下：

ArrayStack.java

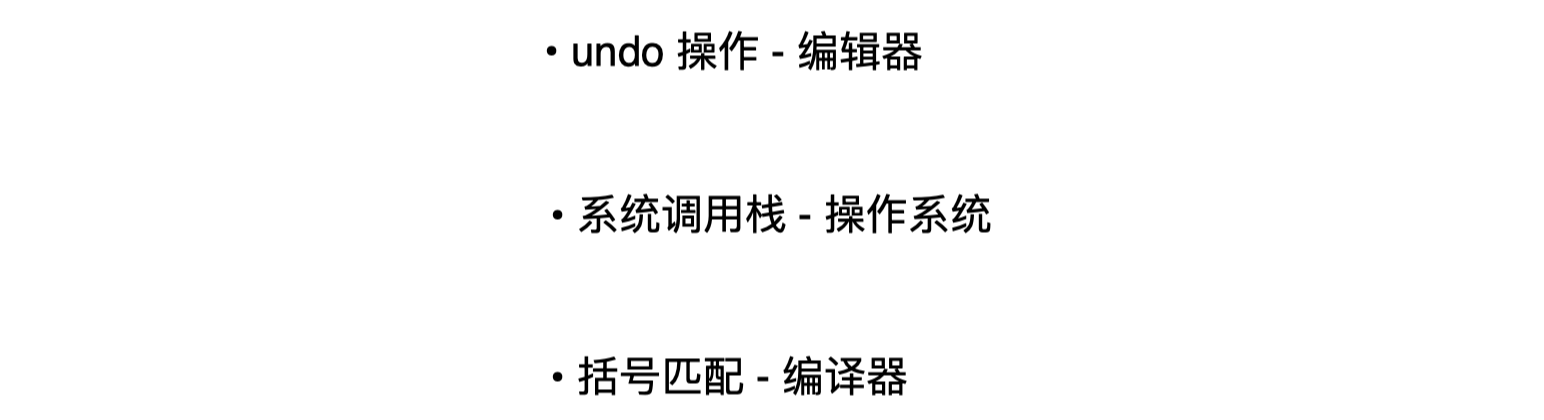
|  |
| --- |
|  |

》》最后对栈的复杂度进行分析一下

A screenshot of a cell phone

Description automatically generated

》》栈的应用



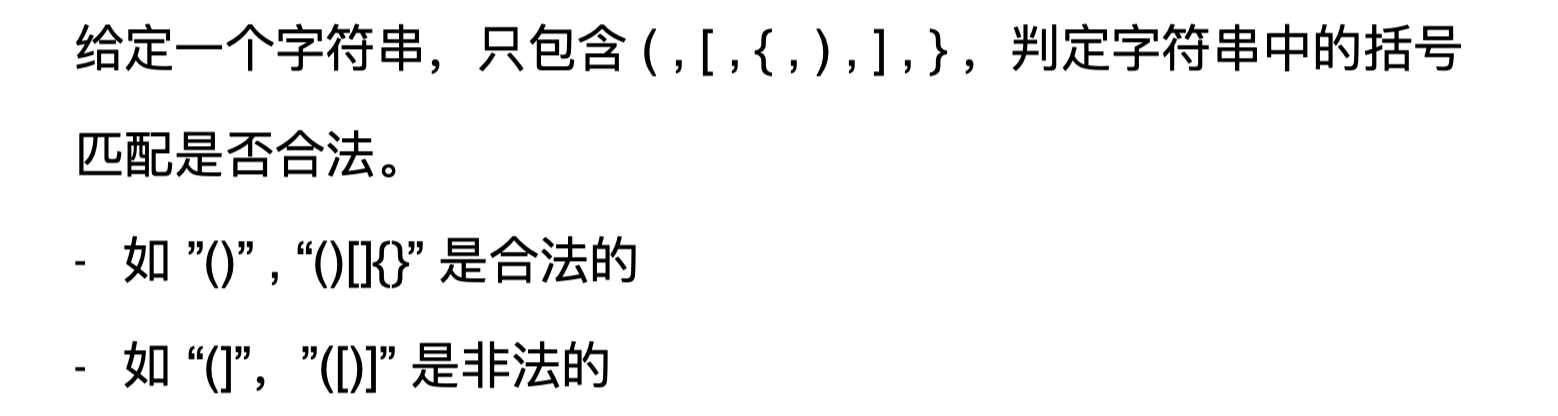
### 栈的另一个应用：括号匹配

》》这里演示 LeetCode 上面的栈的应用，括号匹配

第20 道题

A screenshot of a cell phone

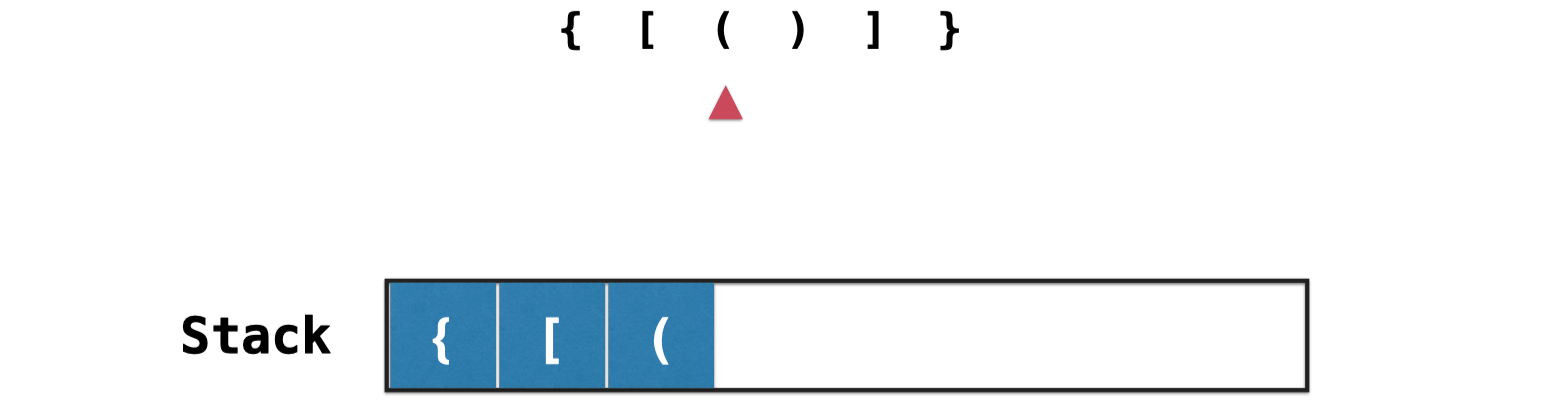
Description automatically generated



》》这里可以通过栈的思路来解决

部分的过程演示截图如下：

这个是匹配的例子：



这个是不匹配的例子：

如果是不匹配，整个过程完后，栈内是有元素的，不为空，表示不匹配

A screenshot of a cell phone

Description automatically generated

》》实践，写代码实现这个括号匹配的问题

这里演示了如何测试自己的Solution，以及测试我们自己的 Stack 类

### 关于 LeetCode 的更多说明

》》更多 LeetCode 上的 stack 的相关说明

》》最后提一下学习方法

A close up of a logo

Description automatically generated

### 数组队列

》》队列的介绍

A screenshot of a cell phone

Description automatically generated

》》队列的特性

A screenshot of a cell phone

Description automatically generated

### 循环队列

### 循环队列的实现

### 数组队列和循环队列的比较

## 最基础的动态数据结构:链表

### 什么是链表

### 在链表中添加元素

### 使用链表的虚拟头结点

### 链表的遍历，查询和修改

### 从链表中删除元素

### 使用链表实现栈

### 带有尾指针的链表：使用链表实现队列

## 链表和递归

### Leetcode 中和链表相关的问题

### 测试自己的 Leetcode 链表

### 递归基础与递归的宏观语意

### 链表的天然递归结构性质

### 递归运行的机制：递归的微观解读

### 递归算法的调试

### 更多和链表相关的问题

## 二分搜索树

### 为什么要研究树结构

### 二分搜索树基础

### 向二分搜索树中添加元素

### 改进添加操作：深入理解递归终止条件

### 二分搜索树的查询操作

### 二分搜索树的前序遍历

### 二分搜索树的中序遍历和后序遍历

### 深入理解二分搜索树的前中后序遍历

### 二分搜索树前序遍历的非递归实现

### 二分搜索树的层序遍历

### 删除二分搜索树的最大元素和最小元素

### 删除二分搜索树的任意元素

### 更多二分搜索树相关话题

## 集合和映射

### 集合基础和基于二分搜索树的集合实现

### 基于链表的集合实现

### 集合类的复杂度分析

### LeetCode 中的集合问题和更多集合相关问题

### 映射基础

### 基于链表的映射实现

### 基于二分搜索树的映射实现

### 映射的复杂度分析和更多映射相关问题

### LeetCode 上更多集合和映射的问题

## 优先队列和堆

### 什么是优先队列

### 堆的基础表示

### 向堆中添加元素和 sift up

### 从堆中取出元素和 sift down

### Heapify 和 Replace

### 基于堆的优先队列

### LeetCode 上优先队列相关问题

### Java 中的 PriorityQueue

### 和堆相关的更多话题和广义队列

## 线段树

### 什么是线段树

### 线段树基础表示

### 创建线段树

### 线段树中的区间查询

### LeetCode 上线段树相关的问题

### 线段树中的更新操作

### 更多线段树相关的话题

## Trie

### 什么是 Trie 字典树

### Trie 字典树基础

### Trie字典树的查询

### Trie 字典树的前缀查询

### Trie 字典树和简单的模式匹配

### Trie 字典树和字符串映射

### 更多和 Trie 字典树相关的话题

### 基于哈希表或者数组的 Trie

## 并查集

### 什么是并查集

### Quick Find

### Quick Union

### 基于 size 的优化

### 基于 rank 的优化

### 路径压缩

### 更多和并查集相关的话题

## AVL

### 平衡树和 AVL

### 计算节点的高度和平衡因子

### 检查二分搜索树性质和平衡树

### 旋转操作的基本原理

### 左旋转和右旋转的实现

### LR 和 RL

### 从 AVL 树中删除元素

### 基于 AVL 树的集合和映射

## 红黑树

### 红黑树与 2-3 树

### 2-3 树的绝对平衡性

### 红黑树与 2-3 树的等价性

### 红黑树的基本性质和复杂度分析

### 保持根节点为黑色和左旋转

### 颜色翻转和右旋转

### 红黑树中添加新元素

### 红黑树的性能测试

### 更多和红黑树相关的话题

### 对于红黑树，任何不平衡多会在三次旋转内解决？

## 哈希表

### 哈希表基础

### 哈希函数的设计

### Java 中 hashCode 方法

### 链地址法 Separate Chaining

### 实现属于我们自己的哈希表

### 哈希表的动态空间处理与复杂度分析

### 哈希表更复杂的动态空间处理方法

### 更多哈希冲突的处理方法

## 结尾语

### 更广阔的数据结构世界，大家加油