# 数据结构及算法基础

数据结构表示数据在计算机中的存储和组织形式，描述元素之间的位置与逻辑关系



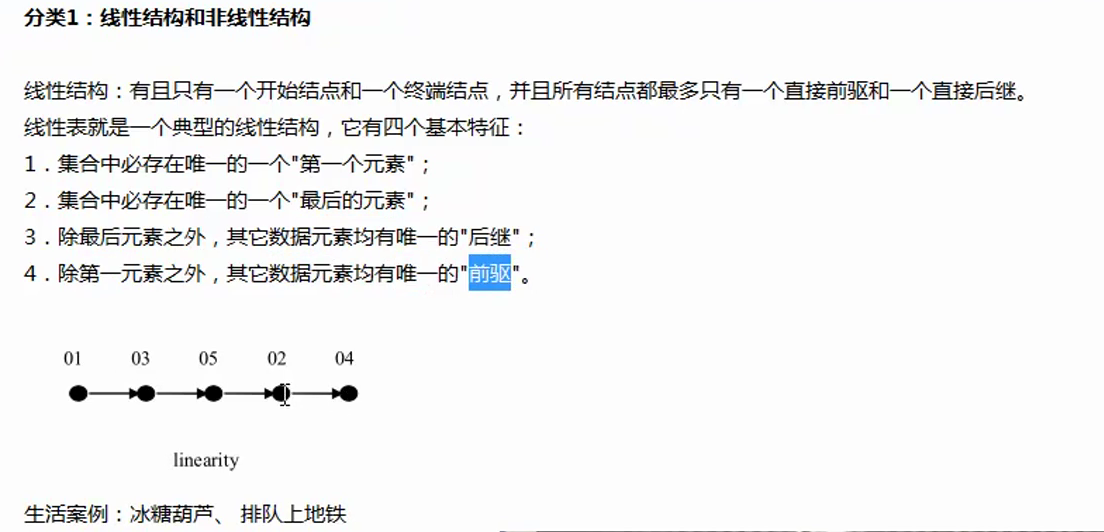
数据结构=逻辑结构+物理结构（存储结构）+各种运算和操作

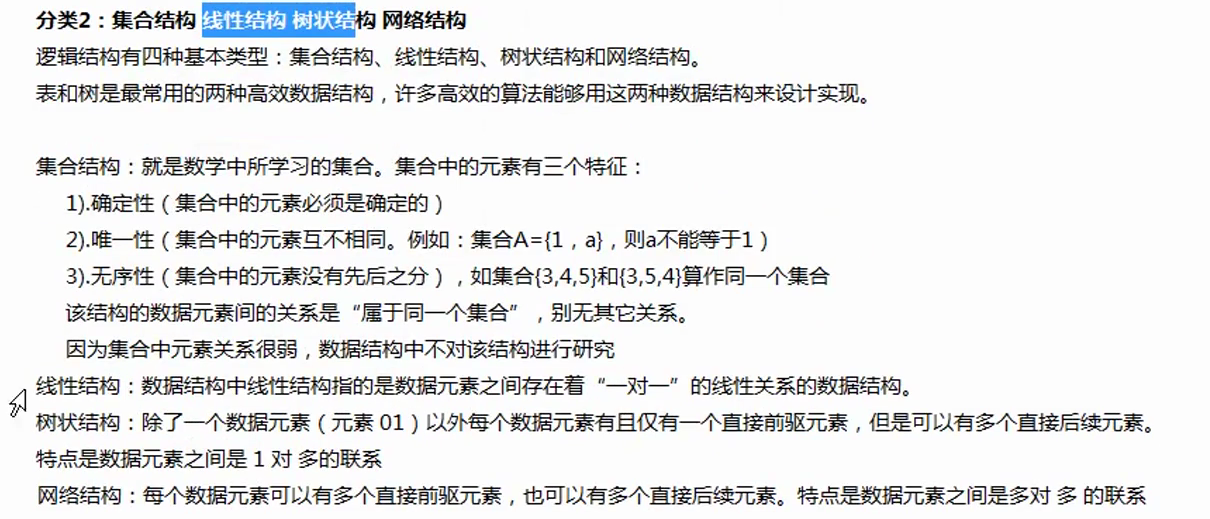
时间复杂度—运行效率

空间复杂度—存储效率

## 逻辑结构

描述数据之间的逻辑关系





1. 集合结构 元素之间同属一个总体不存在其他联系
2. 线性结构 元素之间存在一对一关系
3. 树形结构 元素之间存在一对多关系
4. 网状结构 元素之间存在多对多关系

## 存储结构

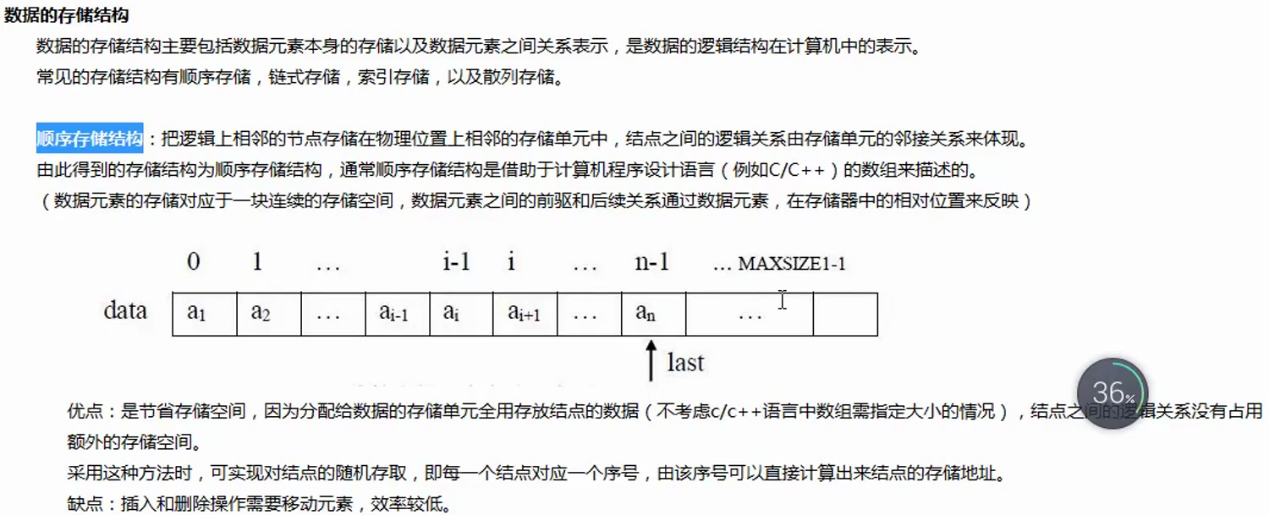
描述元素存储位置关系

1. 顺序结构

逻辑上相邻的两个元素在存储位置上也相邻，通常当数据采用顺序结构存储是需要实现申请一块连续的内存空间（如数组）

优点在于支持随机访问，已知元素下标的情况下访问速度极快

缺点是需要静态分配连续的空间，内存空间的利用率较低，插入元素和删除元素可能需要移动其他元素，效率较低

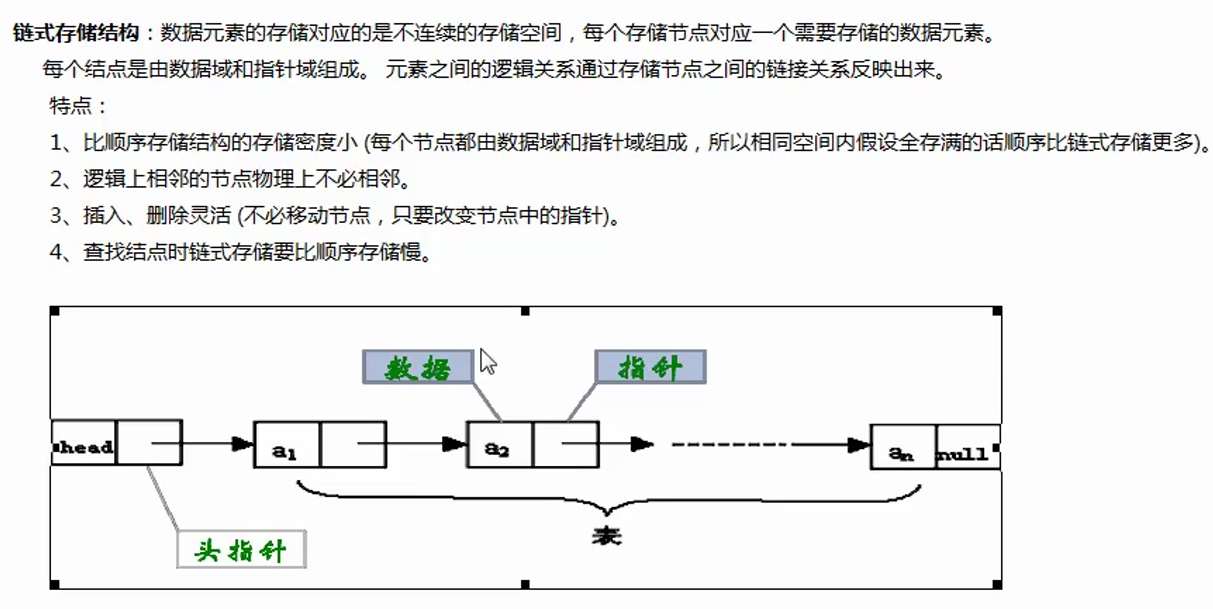


1. 链式结构

逻辑上相邻的两个元素通过指针进行关联，为每一个元素构造一个节点，节点存储元素内容及指针信息指向下一个节点

优点：空间利用率高不需要连续存储空间，插入和删除元素只需要添加和删除指针，增删效率高

缺点：需要额外空间存储指针信息，不支持利用下标对元素随机访问



1. 索引结构

除了存储元素信息之外，单独维护一个索引表存储节点的地址，索引表由若干索引项组成

优点：利用索引号查找元素存储地址，检索速度快

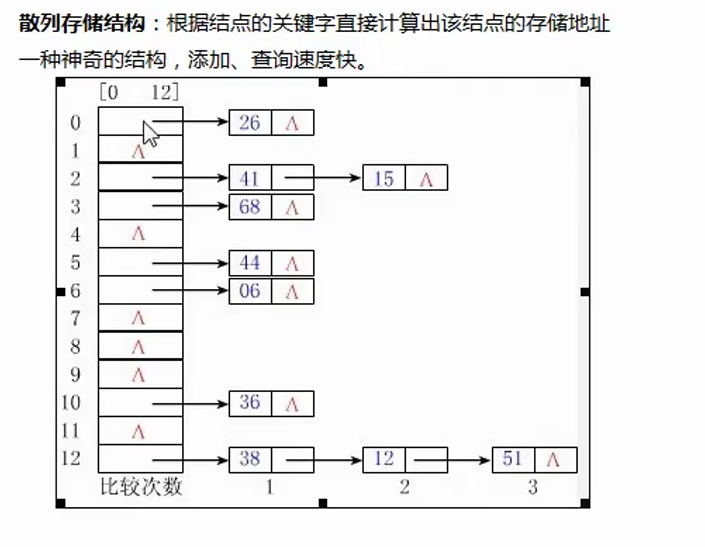
缺点：需要单独空间存储索引表

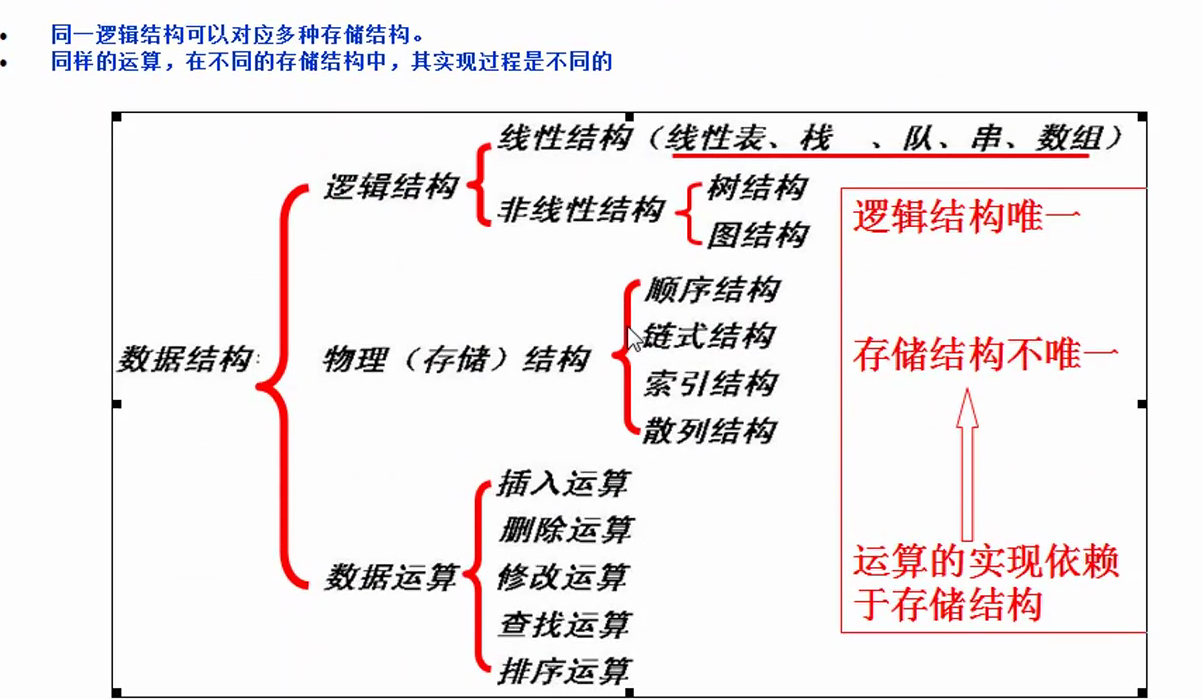
1. 散列结构

通过元素的键通过计算得出存储地址，旨在消除顺序存储结构的插入和删除的低效率与链式存储的检索低效率

优点：散列是数组存储方式的一种发展，采用存储数组中内容的部分元素作为映射函数的输入，映射函数的输出就是存储数据的位置，相比数组，散列的数据访问速度要高于数组

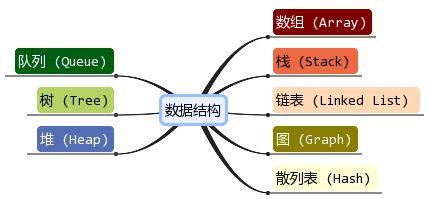
缺点：不支持排序，关键字不能重复，相比线性表存储需要更多空间。

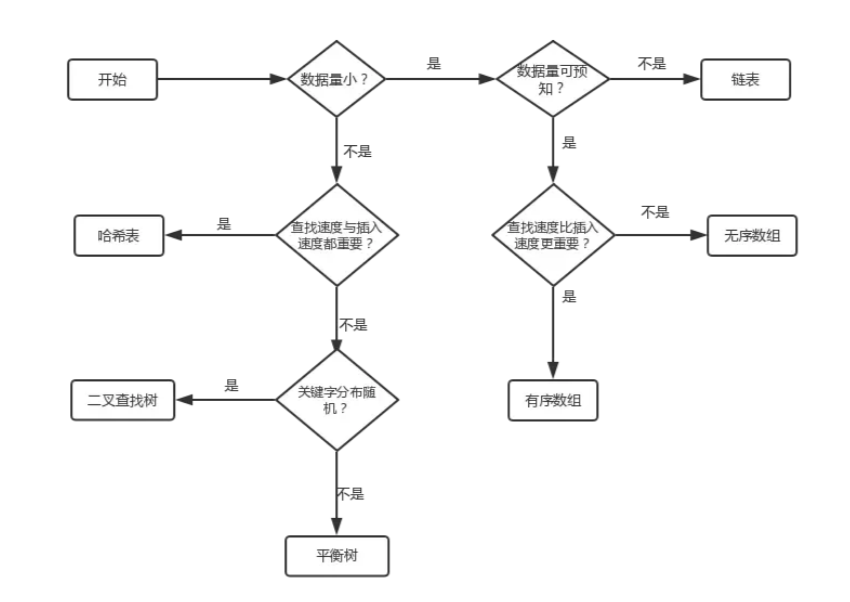




## 数据结构的选择

常见的数据结构

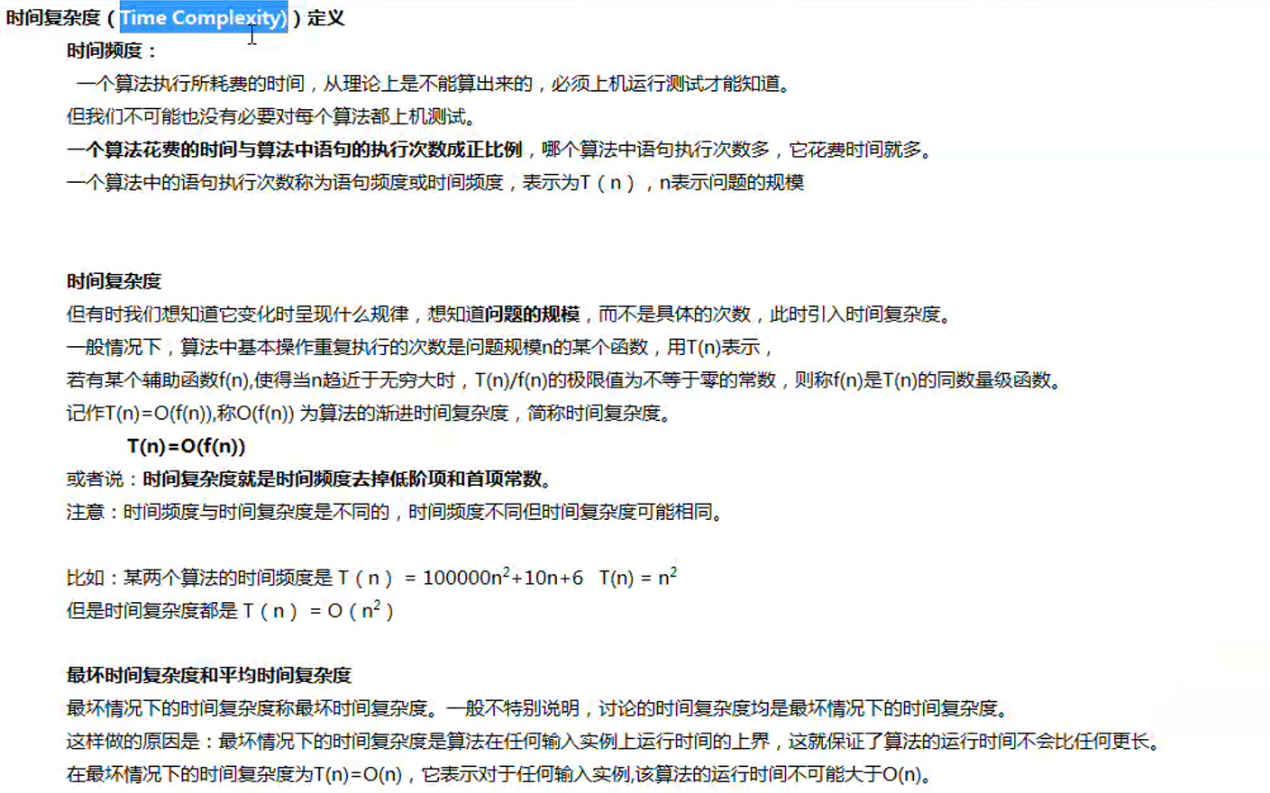


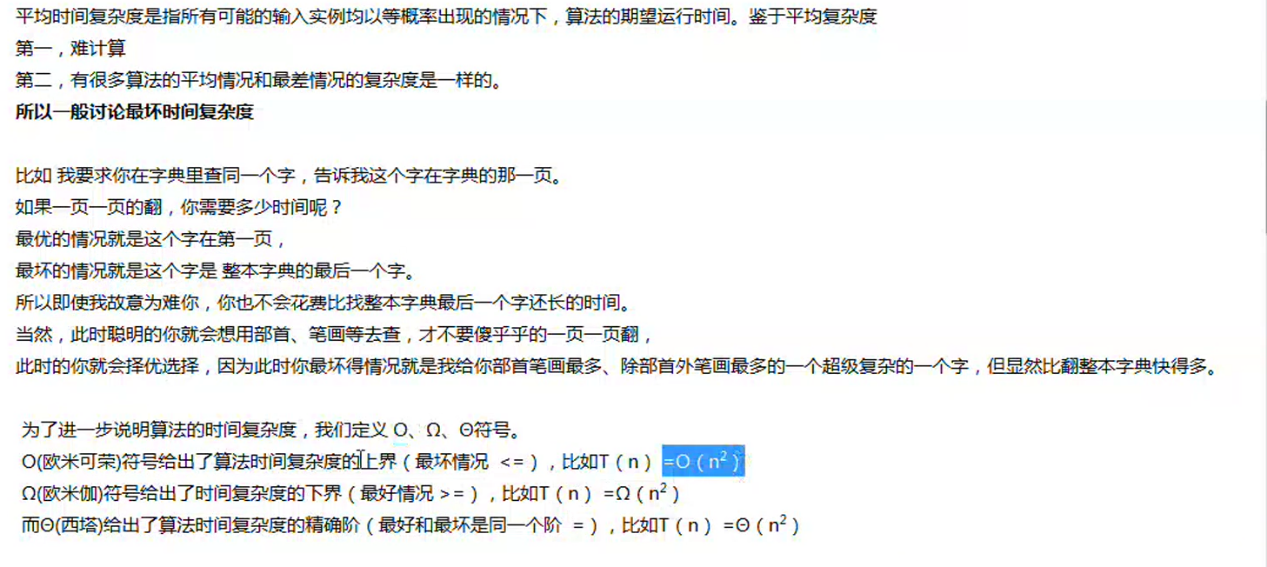


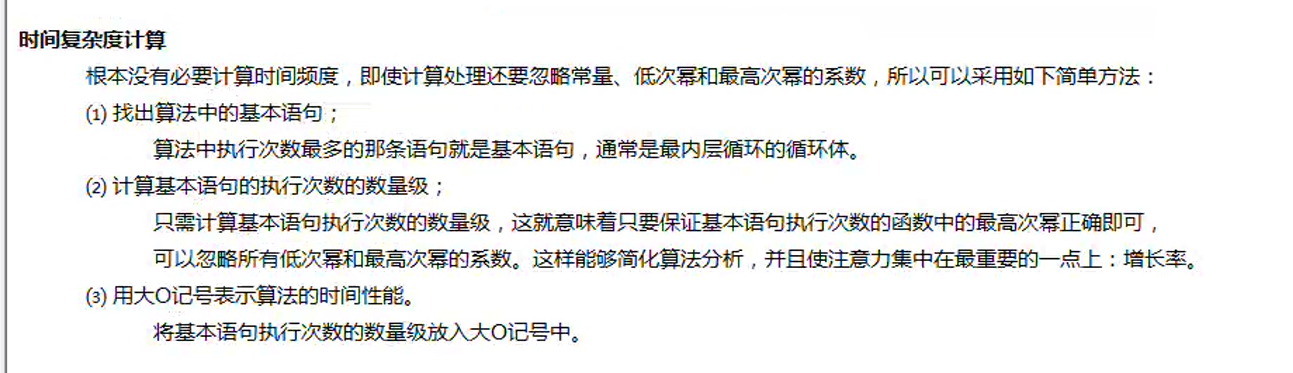
## 算法

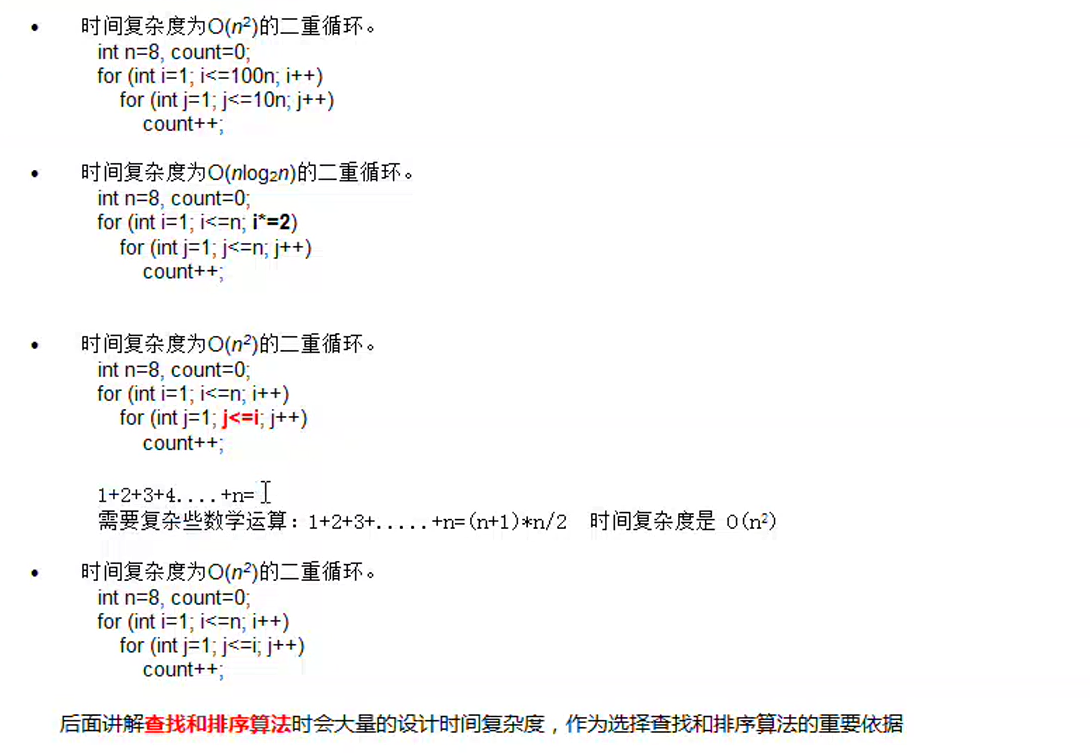


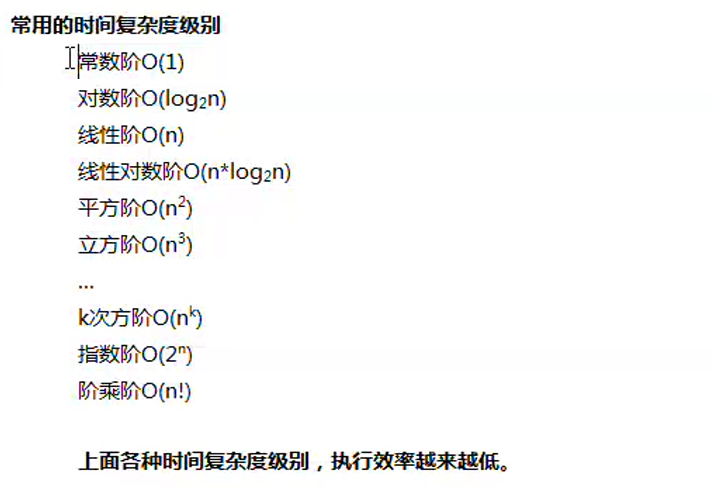
## 时间复杂度

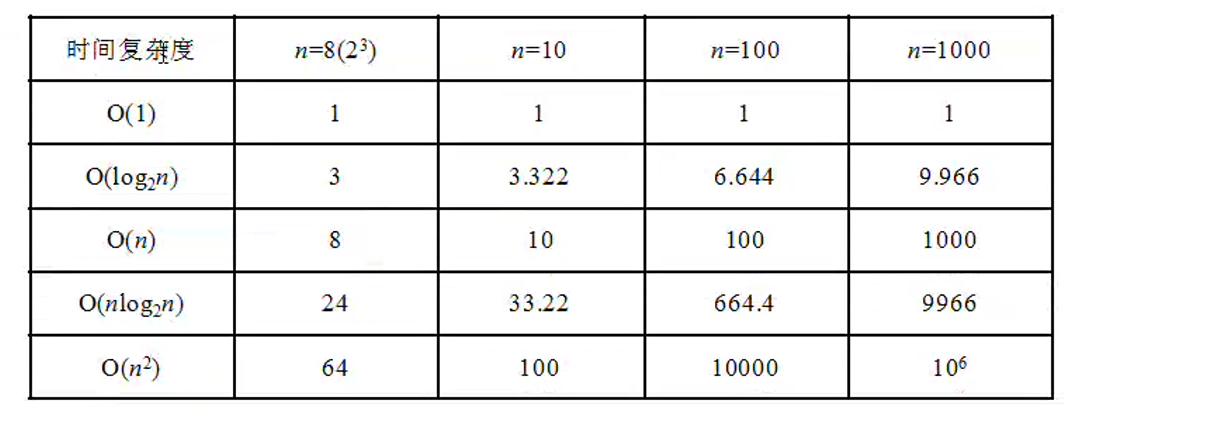


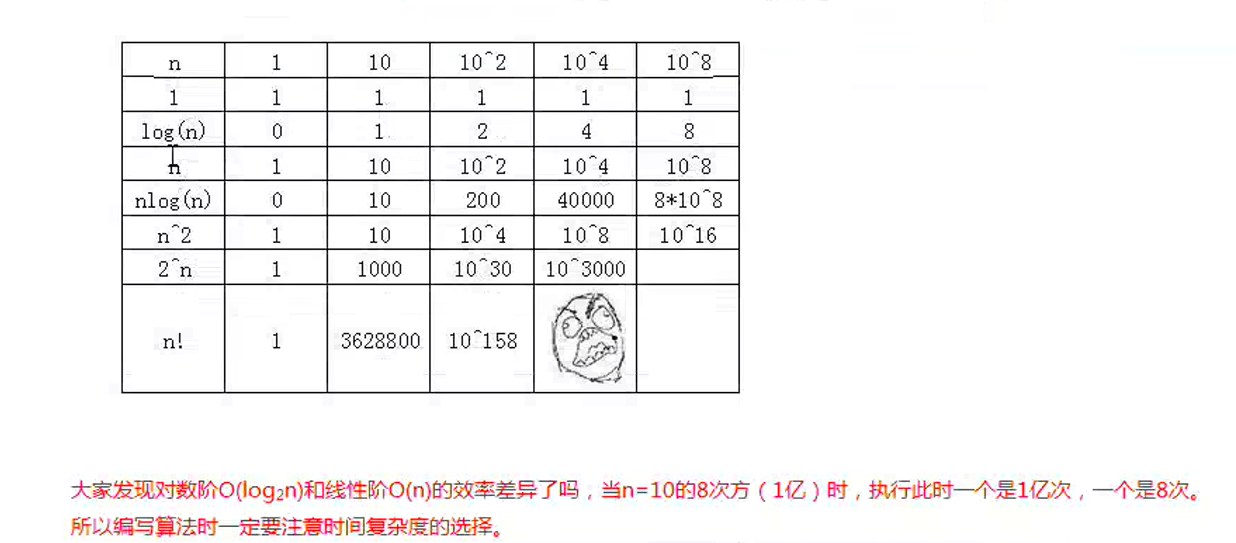




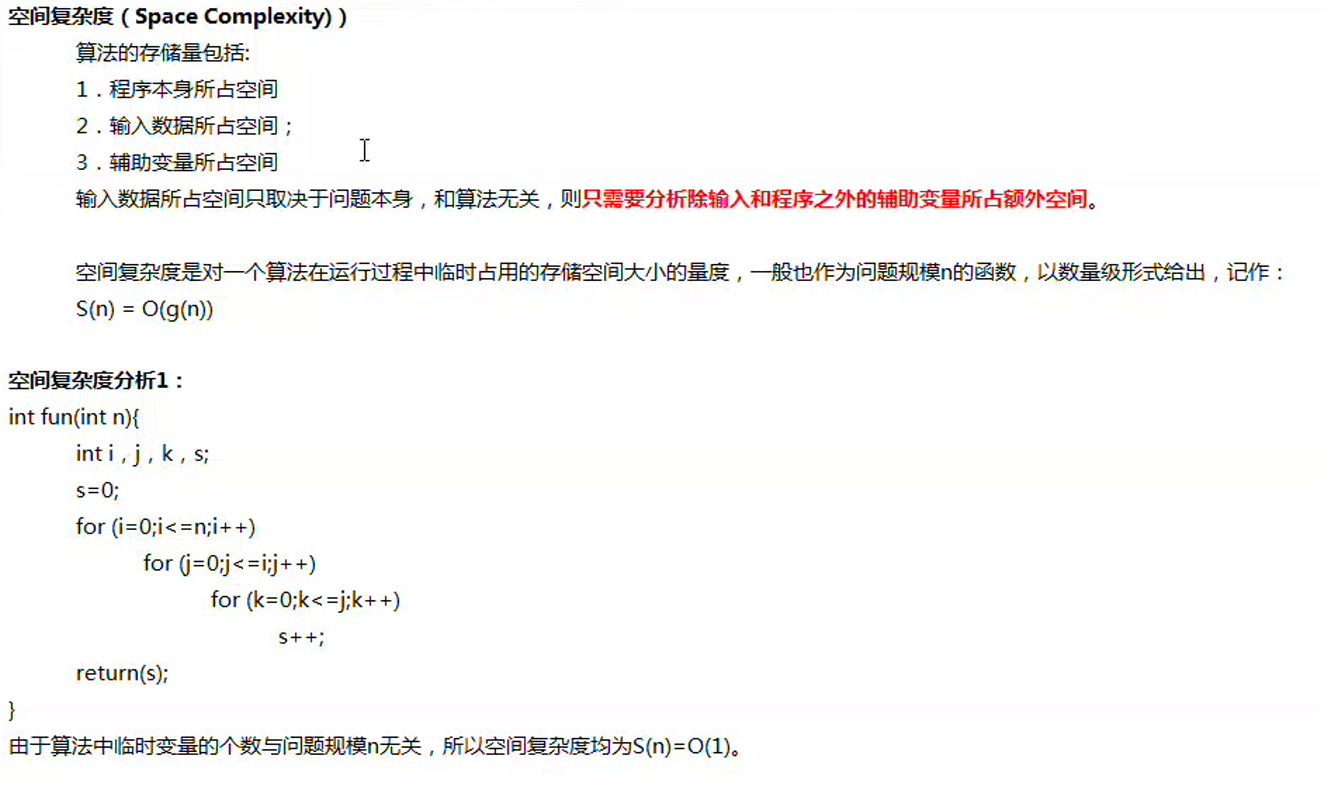


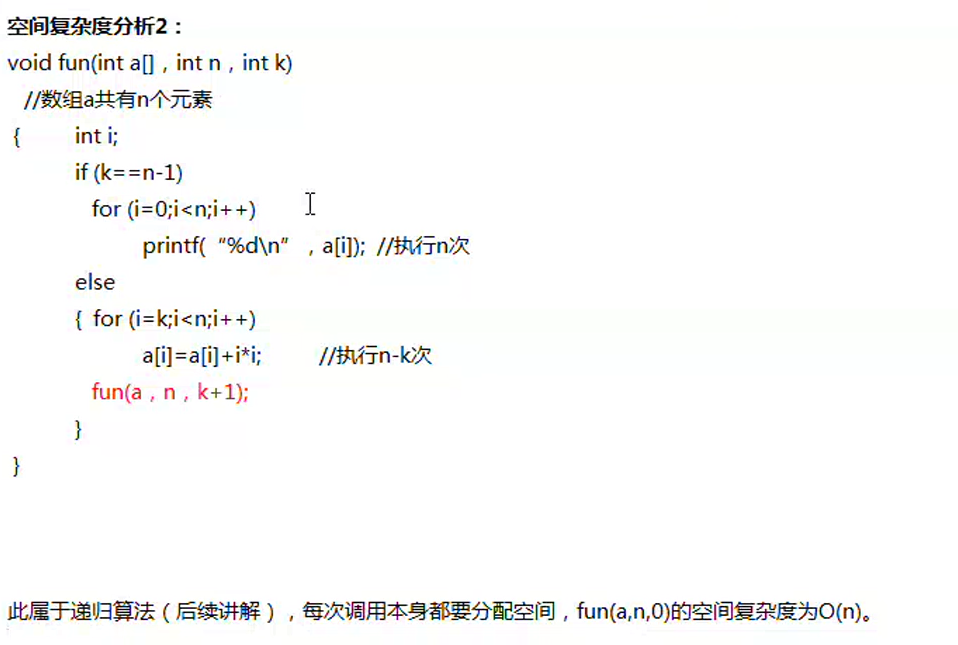






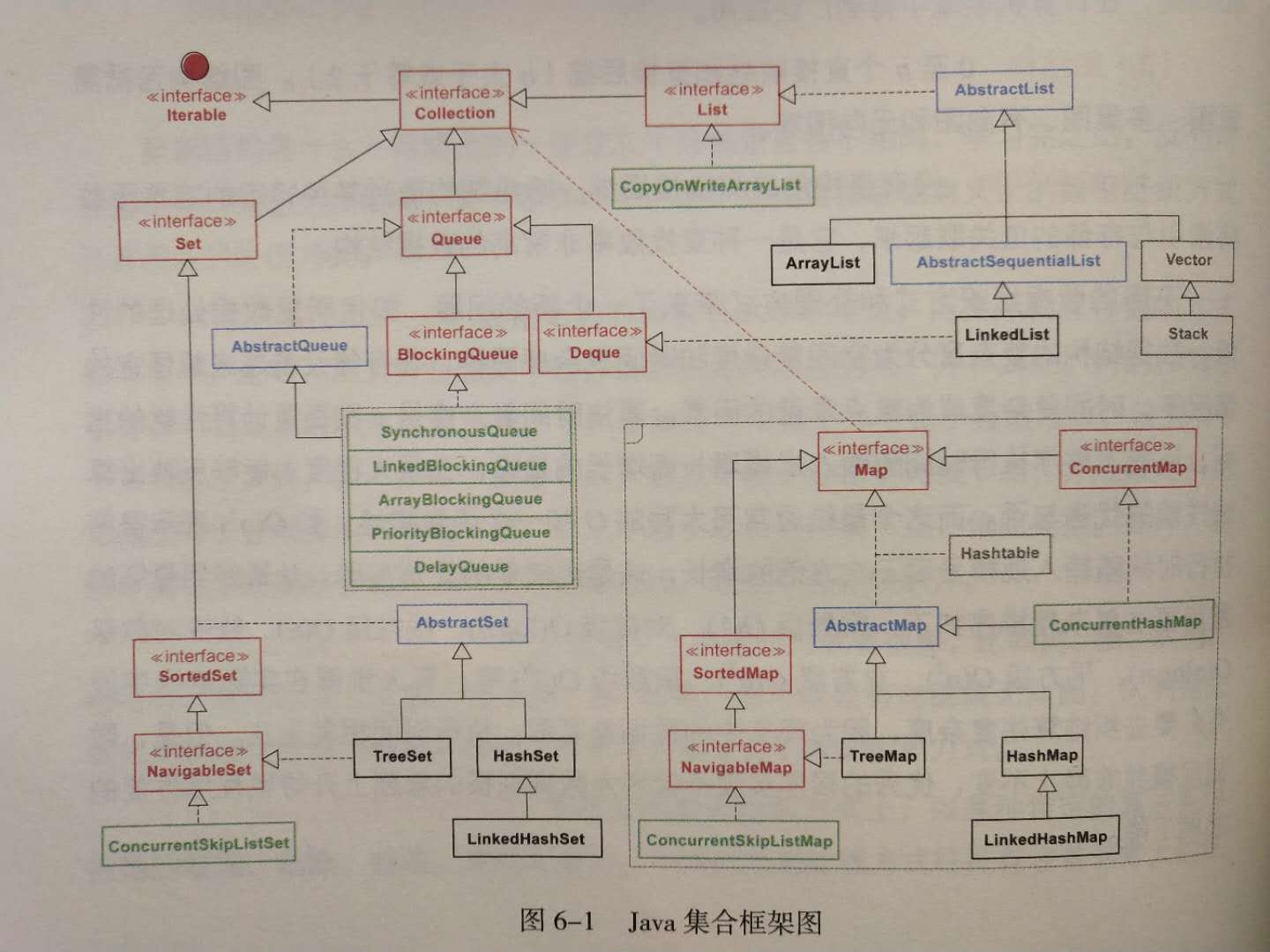
## 空间复杂度





进行到04

# 从数据结构延伸到java集合类



红色代表接口，蓝色代表抽象类，绿色代表并发包中的类，灰色是早期线程安全的类（现基本弃用）

黑色实线代表extends继承,黑色虚线代表implements实现

Java集合实现了常用的数据结构，是数据结构的具体体现。Java集合主要分为两类：一类是按照单个元素存储的collection,在继承树中的set和list都实现了collection接口，另一种是按照key-value存储的map

## List集合

List集合是对线性数据结构的实现，集合中的元素有明确的前驱和后继，也有明确的第一个和最后一个元素，List集合遍历结果稳定，List常用的两个集合类是ArrayList和LinkedList.

ArrayList是容量可以改变的非线程安全的集合，内部使用数组进行存储，扩容的方法是创建更大的数组空间，然后把原有数据复制到新数组中，ArrayList支持对元素的快速随机访问，插入和删除速度很慢，因为有可能涉及大量元素的移动。

默认初始容量是10，最大容量是Integer.max\_value-8 (2147483647-8) 减8是因为Vm会在数组中保存一些头信息预留空间，扩容的原则是 oldSize+(oldSize<<1) ,例如原始容量为13，name扩容后容量为13+6=19 （JDK7及之后）

LinkedList的本质是双向链表，与ArrayList相比，LinkedList插入和删除速度更快，随机访问速度则很慢，十万数据量，随机提取元素速度相差数百倍差距，Linked除了继承AbstractList抽象类之外还实现Deque接口，double-ended queue,同时具有栈和队列的性质。LinkedList具有三个重要成员：size、first、last。LinkedList优点是可以将零散的内存单元通过附加引用的方式关联起来，形成链路顺序查找的线性结构，内存利用率较高（这也是链式存储结构的特点）

Map集合

HashMap

初始化时指定初始容量，为了使hash计算落槽位置更快，应将HashMap容量设置为2的n次幂，如果无法确定初始容量，设置为16.

1. HashMap 默认bucket数组多大？（答案是16）
2. 如果new HashMap<>(19)，bucket数组多大？（答案是32）
3. HashMap 什么时候开辟bucket数组占用内存？（答案是第一次 put 时）
4. HashMap 何时扩容？（答案是put的元素达到容量乘负载因子的时候，默认16\*0.75）
5. HashMap 在 new 后并不会立即分配bucket数组，而是第一次 put 时初始化，类似 ArrayList 在第一次 add 时分配空间。
6. HashMap 的 bucket 数组大小一定是2的幂，如果 new 的时候指定了容量且不是2的幂，实际容量会是最接近(大于)指定容量的2的幂，比如 new HashMap<>(19)，比19大且最接近的2的幂是32，实际容量就是32。
7. HashMap 在 put 的元素数量大于 Capacity \* LoadFactor（默认16 \* 0.75） 之后会进行扩容。（容量(Capacity) 负载因子(Load factor)）
8. JDK8在哈希碰撞的链表长度达到TREEIFY\_THRESHOLD（默认8)后，会把该链表转变成树结构，提高了性能。
9. JDK8在 resize 的时候，通过巧妙的设计，减少了 rehash 的性能消耗。

**存储结构**

JDK7 中的 HashMap 还是采用大家所熟悉的数组+链表的结构来存储数据。

JDK8 中的 HashMap 采用了数组+链表或树的结构来存储数据。