集合相关 数据结构

数组

数组的寻址　头地址+个数×单元大小

从0开始是因为 从1开始多一次减法计算 (i-1)

根据地址获取元素 o(1)

根据（无排序）元素找地址 o(n)

有排序 二分查找 O(logn)

从中间数组插入删除 O(n)　　头尾不一样

Arraylist

底层是动态数组

初始容量为0（不指定容量） 第一次添加数据才会初始化为10

扩容都是原来的1.5倍 每次扩容都要拷贝数组

添加数据时 确保长度够　如果当前数组已使用长度+1后的大于当前数组长度，则扩容
最后返回成功值

> *ArrayList list =new ArrayList(10)*

扩容几次？ 没有扩容 直接new指定容量

> 数组和*List*之间的转换

Arrays.asList转换成list

list.toArray转换成数组

> 转换后 如果原来的修改过 转后的会影响吗？

Arrays.asList会受影响 但list.toArray不受影响

前者源码没有new对象 传入的是引用 相当于包装

后者源码有拷贝到新数组里

linkedlist

单向链表和双向链表

非连续非顺序 存数据和指针

头尾是o(1) 定位是o（n）

> 底层数据结构？和*Arraylist*区别？

Linkedlist是双向链表

查的效率不一样

占用空间不一样

> 两者都不是线程安全的

在方法内使用 局部变量是安全的

或者用Collections.synchronizedList 封装ArrayList 和Linkedlist

hashmap

> 二叉树

二叉搜索树BST　左<根<右

平均插入查找删除时间为O(logn)　　查找最差O(n)

> 红黑树

根节点是黑色

叶子是黑色的空节点

红黑树中红色节点的子节点都是黑色

任一节点到叶子节点的所有路径都包含相同数目的黑色节点

比平衡二叉树快一点 和平衡二叉树一样 防止变成链表　　查找插入删除O(logn)

> 散列表

Hash table　由数组演化过来的 可以根据下标随机访问

用散列函数转为数组下标

散列冲突：拉链法 存链表

插入O(1)

存的比较平均 查找删除是o1　退化成链表为On

用红黑树代替链表 O(logn)

> *Hashmap*底层

Hash表+（链表或红黑树）

链表长度大于8且数组长度大于64 转换为红黑树

1.8开始才用红黑树

> 扩容阈值

扩容阈值=数组容量×加载因子

加载因子为0.75

初始大小为16

hashmap默认懒加载 放东西再初始化

比扩容阈值大就扩容两倍

> 扩容

扩容两倍 旧数组要挪到新数组 哈希有变化

还要判断是不是红黑树和链表 若只有一个节点 则直接挪

将红黑树拆分成2棵子树，如果子树节点数小于等于 UNTREEIFY_THRESHOLD（默认为6），则将子树转换为链表

红黑树节点也是 hash&老容量

如果桶中的元素是链表结构， 遍历 hash&老容量 (位于运算更快) 等于0就位置不变 不等于0新位置变成原索引+oldcap

> *hash方法*

二次哈希 更均匀

`hashCode() ^ (h >>> 16)` 在哈希计算中常见，它用于计算对象的哈希值，目的是为了改进哈希值的分布，使得哈希表中的元素更均匀地分布，减少冲突。

(n-1)&hashcode和 hashcode%n 等价 （**n**是**2**的次方） 位运算更快

> 并发死循环问题

链表是头插法 数据迁移有可能会死循环

1.8改成尾插法就不会死循环了