集合相关 数据结构

Collection是集合接口 Collections是工具类

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

扩容是复制到新数组里

> *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

> 线程安全的*list*

Collecions.synchronizedList

CopyonWriteArrayList 读写锁

Vector

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

Map

for each+entryset方法来遍历

keyset遍历key

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. 泊松分布表明，链表长度达到 8 的概率极低（小于千万分之一）。在绝大多数情况下，链表长度都不会超过 8。阈值设置为 8，可以保证性能和空间效率的平衡。
2. 数组长度阈值 64 同样是经过实践验证的经验值。在小数组中扩容成本低，优先扩容可以避免过早引入红黑树。数组大小达到 64 时，冲突概率较高，此时红黑树的性能优势开始显现。

如果在红黑树结构下的元素数量下降到 **6** 以下，则会从红黑树重新转回链表。 没其他条件

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 （位移就行了 更快）

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 哈希值的高16位和低16位混合，生成新的哈希值

(h = key.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。

然后是(n - 1) & hash 计算桶位置

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

key可以为**Null**

> 怎么查？*hashcode*和*equals*的问题

`HashMap` 会通过键的 `hashCode()` 来决定该键应该存储在哪个桶中

`equals()` 用来判断两个对象是否相等。如果两个键的 `hashCode()` 相同（即发生了哈希冲突），`HashMap` 会使用 `equals()` 来判断这两个键是否真正相等。如果相等，则认为它们是同一个键。

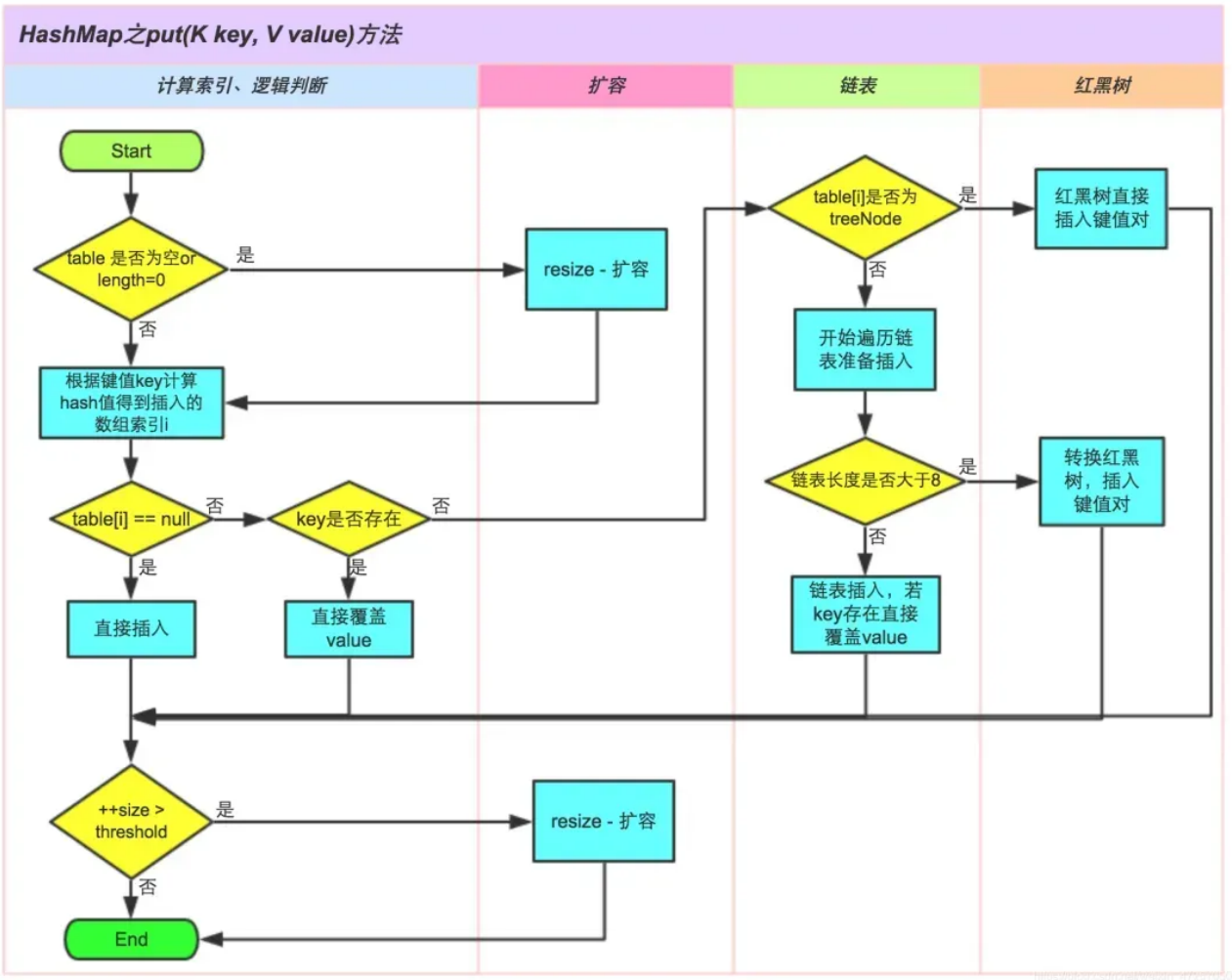
equals相等则hashCode一定相等　hashCode相等 equals不一定相等

> 并发问题

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 但是有可能覆盖原元素

> *put*过程过程？



Hasttable

HashTable也线程安全 内部方法都被synchronized修饰了

`HashMap` 对哈希值进行了高位和低位的混合扰动处理以减少冲突，而 `Hashtable` 直接使用键的 `hashCode()` 值。

已经弃用了

Set

hashset实现

依赖于 `HashMap` 来实现的。每个 `HashSet` 元素都作为 `HashMap` 的 **key** 存储，而 `value` 是一个固定的 占位对象。

先计算 `key` 的 **hashCode**，确定在 `HashMap` 数组中的索引位置。

进行 `equals()` 判断，如果 `equals()` 认为相等，则不插入，否则插入到链表或红黑树中。

优先队列和队列

`Queue` 和 `Deque` 只是接口，具体的底层实现依赖于其常见的实现类，

Deque 继承了 Queue

> 实现类有哪些？

`ArrayDeque` 是基于可变长的数组和双指针来实现，

`LinkedList` 则通过链表来实现。（`LinkedList` 既实现了 `Queue`，也实现了 `Deque`。)

`PriorityQueue`（堆实现）

线程安全队列：`ArrayBlockingQueue` 数组（环形队列）+ 可重入锁

> *PriorityQueue*？

是非线程安全的，且不支持存储 `NULL` 和 `non-comparable` 的对象。

`PriorityQueue` 是 基于最小堆（**Binary Heap**） 实现的优先级队列，它不是 **FIFO** 队列，而是按照元素的优先级 进行排列，每次 `poll()` 操作都会返回当前队列中 优先级最高（最小的） 的元素。

二叉堆 可变长数组

`PriorityQueue` 默认是最小堆（即堆顶是最小值），但可以通过 **Comparator** 变成 最大堆（即堆顶是最大值）。

杂

HashMap和HashTable分别使用containsKey()或containsValue()方法来判断是否包含某个key或value,而不是contains方法