六、Map 集合 （映射）

1、Map集合特点

主要用于存储key-value格式的元素

不能保证元素的添加顺序，

其主要实现类:HashMap,TreeMap

2、HashMap

底层通过Hash表来实现

1>HashMap特点

1)不能保证元素的添加顺序

2)经过equals比较key相同，value值覆盖

3)key或value值为null

如果key值为null，会使用==比较key

2>对象创建

1>无参构造方法

```
HashMap<K,V> map = new HashMap<>();
```

3>主要方法

1)put(K,V) 存储键值对

2)get(K) 根据key获取value值

3)remove(K) 根据key移除键值对

4)containsKey(K) 判断是否包含指定key值

5)clear() 清空集合中所有的键值对

6)size() 获取集合中键值对个数

7)isEmpty 判断是否为空集合

4>map集合的三种遍历方式

1)遍历所有的key

```
Set<K> set = map.KeySet();
```

2)遍历所有的value

```
Collection<V> values = map.values();
```

3)遍历所有的key-value

```
Set<Entry<k,V>> set = map.entrySet();
```

一个key-value键值对本质上就是一个Entry对象，

所以Map集合中其实装的是多个Entry对象

5>HashMap底层实现原理（面试重点）

HashMap底层实现为哈希表（散列表）

Java集合中哈希表的实现方案:

Jdk<=1.7:哈希表= 数组+单向链表

jdk>=1.8:哈希表= 数组+单向链表+红黑树

集合默认长度为16，加载因子：0.75（系数）

单向列表转红黑树 8个

红黑树转单向列表 6个

5.1>数组

当使用无参构造器创建HashMap对象，

默认初始化加载因子为0.75，当第一次调用

put()方法向集合中添加元素时，会初始化一个

长度为16的Node[]数组，Node代表一个键值对结点对象，

当数组容量达到阈值，也就是数组长度*加载因子时，

如：第一次为16*0.75个时。数组就会进行扩容，每次扩容之前的二倍

然后将之前数组中所以结点取出来，

根据Key重写结算新的哈希值和下标值，再存入到新的数组中。

5.2哈希表数据具体的存取过程

5.2.1>存储数据

1)根据key调用哈希算法计算出一个哈希值，

然后再根据哈希值计算出一个下标值

2)如果该下标位置没有任何结点，呢么就直接存储

3)如果该下标位置有其他结点

3.1)该下标位置有且仅有一个结点，呢么就调用equals()方法

来比较两个key值

3.1.1）如果经过equals()比较返回true，说明是相同key，

就覆盖该节点

3.1.2)如果经过equals()比较返回false，呢么说明只是

正好算出来的下标值相同，但不是重复的key，

呢么就以单向链表的方式挂再一起

3.2）如果该位置已经是一个单向链表或者红黑数，

呢么就遍历该链表或红黑树上的结点，

依次取出结点的key和要插入的结点的key进行equals()比较

3.2.1）如果某一个结点经过equals()比较为true，
呢么就覆盖该位置结点，后续不再比较
3.2.2)如果使用结点经过equals()比较都返回false，
呢么就继续插入到链表的末尾。
3.2.3)在Jdk1.8之后，为了提高查询效率，减少比较次数，
当某一个单向链表上的结点个数超过8个，
并且数组长度超过64个单向链表就会变成一颗红黑树.
但是如果只是单向链表结点超过8个，但是数组长度没有超过64
呢么就对数组进行翻倍扩容。然后将之前数组中所以
节点取出来，根据key重新计算新的哈希值和下标值，
再存入到新的数组中
3.2.4)当红黑树上的节点个数减少6个以下时，
红黑树又会变回为单向链表
3.2.5）当数组扩容后，原数组中数据插入到新数组后，单向链表
的插入方式
jdk<=1.7:头插发
jdk>=1.8:尾插法（避免多线程死链）

面试题：

1)equals()和hashCode()两个方法的调用顺序以及何时调用

2)两个对象的hash值相等，呢么equals比较一定相等么?

3)两个对象的equals()比较相等，哈希值一定相等?

4)为什么equals()比较相等，为什么哈希值也相等

5)为什么equals()和hashCode()两个方法必须要同时重写

6>二叉排序树（二叉查找树，二叉搜索树）

前序遍历

57-36-23-15-42-88-70-90-100

中序遍历

15-23-36-42-57-70-88-90-100

后序遍历

15-23-42-36-70-100-90-88-57

特点：

1>每个夫结点最多有两个子节点，左子节点不为空一定小于他的夫结点，右子节点不为空一定大于他的父节点。

2>如果层数i从0开始，第i层最多有2^i个结点

3>查找采用二分查找算法，最大查找次数等于二叉树的高度，查找效率高，时间复杂度O(ln)

4>结点内容不能为null，且结点之间值不能相同（equals），且必须能够比较大小（Comparable,Comparator）。

5>遍历方式：

5.1>前序遍历（夫->左->右）

5.2>中序遍历（左->夫->右）

5.3>后序遍历（左->右->夫）

6>三种二叉树

6.1>完美二叉树

除了叶子结点之外的每一个结点都有两个孩子，每一层(当然包含最后一层)都被完全填充。

6.2>完全二叉树

除了最后一层之外的其他每一层都被完全填充，并且所有节点都保持向左对齐。

6.3>完满二叉树

除了叶子节点之外的每一个节点都有两个孩子节点，要么没有子节点，要么有两个子节点

7>红黑树

往二叉排序树中插入数据时，如果后面数据一直比前面数据小或比前面数据大，呢么二叉排序树就会坍塌成一个单向链表，失去了平衡性，查找效率就会下降，所以需要就使用红黑树来保证二叉树在插入数据时的平衡性，红黑树本质上是一个能够实现自平衡的二叉排序树。

7.1>红黑树特点

1)每个结点要么是红色，要么是黑色

新插入的结点默认是红色的

2)根节点必须是黑色的

3)每个红色结点的两个子节点必须是黑色结点

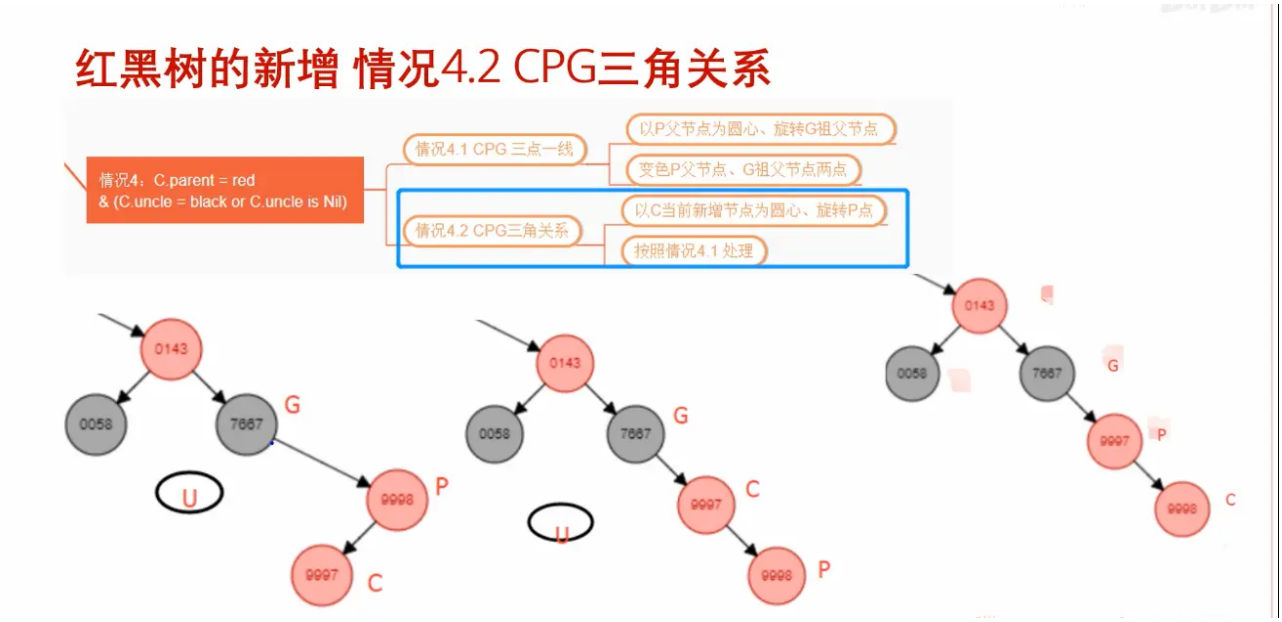
4)每个叶子节点都是黑色的空节点（值为null）

5)从任一节点到每个叶子节点都包含相同数量的黑色节点

7.2>红黑树如何实现自平衡

1)变色(红色->黑色，黑色->红色)

2)旋转(左旋，右旋，旋转中心点，方向，旋转节点)



8>Hashtable（哈希表）

与HashMap和Hashtable有什么区别

1)都是Map接口的实现类，都可以存储键值对

内部方法基本都相同

2)HashMap是线程不安全的

Hashtable是线程安全的

3)HashMap允许key或vaue为null，

Hashtable不允许key或value为null

4)HashMap和Hashtable底层实现都是哈希表

HashMap 哈希表=数组+单向链表+红黑树

Hashtable 哈希表=数组+单向链表

5)HashMap是在第一次调用put方法时才会初始化16个长度的数组（懒加载）

Hashtable是在创建对象时就会立即初始化11个长度的数组（立即加载）

9>Properties 性能

1、是Hashtable的子类，可以用于读取后缀为.properties的属性配置文件

2、主要方法

1)load() 通过流读取属性配置文件

2)getProperty() 根据key获取value

3、TreeMap

（主要是可以对entry进行排序输出，可以自己制定比较规则。）

1)底层将数据存储到红黑树中，不保证元素的添加顺序，但是可以根据key来对元素进行排序，如果两个key经过比较后大小相等，那么value值会覆盖，key不允许为null。

2)对象创建

2.1)无参构造器创建对象

treeMap<K,V> map = new TreeMap<>();

要求K的类型上必须实现Comparable可比较接口。

2.2)外接比较器创建对象

treeMap<K,V> map = new TreeMap<>(Comparator cmp);

以Comparator比较器制定的规则对key进行排序。

3)主要方法

参考HashMap方法

七、Set集合（设置）

Set接口是继承自Collection的子接口，用于存储单个元素，

存储在该集合中的元素是不允许重复的。

也不能保证元素的添加顺序，元素也没有下标。

主要实现类：HashSet,TreeSet

1、HashSet底层实现原理

底层是将数据存储到HashMap中，其中添加的元素会被当做是一个key值，value值是一个没有意义的Object对象，要求添加进HashSet集合中的元素必须重写hashCode()和equals()两个方法，如果这两个方法比较都相等，那么就会被认为是重复元素，后面的元素就不会继续存储了

2、对象创建

HashSet<E> set = new HashSet<>();

底层自动构建一个HashMap

3、主要方法

参考ArrayList，因为实现是哈希表，所以没有根据下标操作元素的方法。

2、TreeSet

1)底层是将数据存储到TreeMap中，其中添加的元素会被当做是一个key值，value值是一个没有意义的Object对象。

不保证元素的添加顺序，但是能够对元素进行排序，所以要求元素之间必须能够比较大小，且不能为null，如果两个元素之间经过比较大小相等，则认为是重复元素，那么后面的元素就不存储了。

2)创建对象

2.1>无参构造

TreeSet<E> treeSet = new TreeSet<>();

E要求必须实现Comparable可比较接口，内部会创建一个TreeMap对象。

2.2>外接比较器创建对象

TreeSet<E> set = new TreeSet<>(Comparator cmp)

按照比较器制定的规则对元素进行排序

2.3>主要方法

参考ArrayList，因为底层实现为红黑树，所以没有根据下标操作元素的方法。

八、Stream流式操作（了解）

1、什么是Stream流

将要处理的集合或数组数据中元素看做是一种流，借助Stream API和lambda表达式来对流中元素进行各种操作。

2、Stream流的操作步骤

1)获取Stream流对象，并将集合或数组中的数据放到流上

1.1）Collection获取流对象

借助Collection接口中的stream()默认方法。

Stream<E> stream = 集合.stream();

1.2)Map集合获取流对象

Map集合无法直接获取流对象，只能获取key或者value或entry对象的流。

1>获取key的流

Stream<E> stream = map.keySet.stream();

2>获取value的流

Stream<E> stream = map.values().stream();

3>获取key-value的流

Stream<Entry<K,V>> stream = map.entrySet().stream();

1.3)数组获取流对象

借助数组工具类Arrats.stream()静态方法。

Stream<E> stream = Arrays.stream(arr);

1.4)一堆零散数据获取流对象

借助Stream接口的of()静态方法。

Stream<E> stream = Stream.of(T... values);

3、Stream流的中间操作方法

中间方法可以连续调用多次，每次都会返回一个新的流

1)filter() 过滤器，根据条件过滤流中的内容

2)limit(int n) 获取前n个元素

3)skip(long n) 跳过n个元素

4)distinct() 元素去重（经过equals比较相等）

5)concat() 合并两个流为一个流

6)map() 转换流中数据

4、最终操作

每个流只能操作一次，操作完这个流就结束了，不能再调用其他的方法了。

1)forEach() 迭代遍历

2)count() 统计流中元素的个数

3)toArray() 收集流中剩余数据到数组中

4)collet() 收集流中剩余数据到集合中

4.1)collect(Collectors.toList)

收集到list集合中

4.2)collect(Collectors.toSet)

收集到set集合中

4.3)collect(Collectors.toMap(keyFunction,valueFunction))

收集到map集合中