目录

[1、 ArrayList 3](#_Toc7991)

[1.1、 空参构造ArrayList() 3](#_Toc19685)

[1.2、 构造函数ArrayList(Collection<? extends E> c) 3](#_Toc9974)

[1.3、 构造函数ArrayList(int) 3](#_Toc8357)

[1.4、 boolean add(E) 3](#_Toc18534)

[1.5、 Void add(int index, E element) 4](#_Toc19813)

[1.6、 boolean addAll(Collection<? extends E> c) 4](#_Toc22926)

[1.7、 boolean addAll(int index, Collection<? extends E> c) 4](#_Toc29649)

[1.8、 Void clear() 5](#_Toc30787)

[1.9、 Object clone() 5](#_Toc755)

[1.10、 boolean contains(Object o) 5](#_Toc9664)

[1.11、 void ensureCapacity(int minCapacity) 5](#_Toc5101)

[1.12、 E get(int index) 5](#_Toc10260)

[1.13、 int indexOf(Object o) 6](#_Toc26692)

[1.14、 Iterator<E> iterator() 6](#_Toc10167)

[1.15、int lastIndexOf(Object o) 7](#_Toc18313)

[1.16、ListIterator<E> listIterator() 7](#_Toc29444)

[1.17、ListIterator<E> listIterator(int) 8](#_Toc8051)

[1.18、E remove(int index) 8](#_Toc24183)

[1.19、boolean remove(Object o) 8](#_Toc8612)

[1.20、boolean removeAll(Collection<?> c) 8](#_Toc12251)

[1.21、void sort(Comparator<? super E> c) 9](#_Toc13486)

[1.22、List<E> subList(int fromIndex, int toIndex) 9](#_Toc13251)

[1.23、Object[] toArray() 9](#_Toc31242)

[1.24、T[] toArray(T[] a) 9](#_Toc22541)

[1、 Vcetor 9](#_Toc31023)

[1、 Stack 10](#_Toc15166)

[2、 LinkedList 10](#_Toc16353)

[2.1、 构造函数LinkedList() 10](#_Toc24566)

[2.2、 构造函数LinkedList(Collection<? extends E> c) 10](#_Toc2669)

[2.3、 boolean add(E e) 10](#_Toc27856)

[2.4、 void clear() 10](#_Toc23166)

[2.5、 Object clone() 10](#_Toc25700)

[2.6、 boolean contains(Object o) 11](#_Toc22932)

[2.7、 Iterator<E> descendingIterator() 11](#_Toc24139)

[3、 HashSet 11](#_Toc3197)

[4、 LinkedHashSet 11](#_Toc20822)

[5、 11](#_Toc18227)

[6、 HashMap 11](#_Toc16463)

[6.1、构造函数HashMap() 11](#_Toc3344)

[6.2、构造函数HashMap(int initialCapacity) 12](#_Toc26115)

[6.3、构造函数HashMap(int initialCapacity, float loadFactor) 12](#_Toc10037)

[6.3、构造函数HashMap(Map<? extends K, ? extends V> m) 12](#_Toc8811)

[6.4、V put(K key, V value) 12](#_Toc19351)

[6.5、void clear() 13](#_Toc14878)

[6.6、Object clone() 13](#_Toc16659)

[6.7、boolean containsKey(Object key) 14](#_Toc17028)

[6.8、Set<Map.Entry<K,V>> entrySet 14](#_Toc4465)

[7、 LinkedHashMap 14](#_Toc10329)

[7.1、LinkedHashMap() 14](#_Toc8264)

[7.2、LinkedHashMap(int initialCapacity) 14](#_Toc27515)

[7.3、LinkedHashMap(int initialCapacity, float loadFactor) 14](#_Toc30226)

[8、 TreeMap 15](#_Toc16936)

[8.1、TreeMap() 15](#_Toc18692)

[8.2、TreeMap(Comparator<? super K> comparator) 15](#_Toc9994)

[8.5、V put(K key, V value) 15](#_Toc22870)

[8.6、boolean containsValue(Object value) 16](#_Toc12487)

## ArrayList

### 空参构造ArrayList()

调用此构造函数后，将一个没有成员的数据赋值给成员变量elementData

### 构造函数ArrayList(Collection<? extends E> c)

先将参数集合转换为数组，如果数组的长度为0，则将EMPTY\_ELEMENTDATA赋值给elementData

### 构造函数ArrayList(int)

根据参数，创建指定容量的数组

### boolean add(E)

说明：

elementData -集合底层存储数据的数组

DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA-默认空的object对象数组

Size-集合中存储的数据的数量

modCound-集合结构被修改的次数（比如修改某一索引下的引用，就不包括在内）

newCapacity-elementData 数组扩充后的容量

方法作用：将指定的元素追加到此列表的末尾。

①首先，如果elementData == DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA，则取出 DEFAULT\_CAPACITY与size+1中较大的值，否则取出size+1；

②然后modCount自增长1；

③然后判断第①步中取出的值是否大于elementData 数组的长度，如果是，则进行 elementData 数组扩充

④elementData 数组的新容量newCapacity=老容量+老容量右移1位，并判断新容量是 否超出int类型最大值，如果是，则新容量等于int类型最大值；

⑤将老数组中的数据复制到新数组中，并将新数组赋值给elementData ；

⑥最后在elementData [size+1]处插入本次新增数据。

注意：

①通过无参构造创建的集合，在首次添加数据时，elementData 数组容量会设置为10

②当本次需要插入数据的索引值大于elementData 数组容量时，会对elementData 进行 扩容，每次的扩容量为elementData 容量的一半

③添加的对象可以为空，也可以重复添加同一对象

### Void add(int index, E element)

方法作用：在此列表中的指定位置插入指定的元素。

①检查插入的索引位置index是否在0到size之间，如果不是，则抛出索引越界异常；

②modCount自增长1，检查elementData 数组是否需要扩容，需要扩充则扩充；

③将elementData 数组index到size-1索引位置的对象复制到index+1到size位置上， 此操作是调用System.arraycopy()方法完成；

④最后在index插入本次要插入的对象，size自增1.

### boolean addAll(Collection<? extends E> c)

方法作用：按指定集合的Iterator返回的顺序将指定集合中的所有元素追加到此列表的 末尾。

①将传入的集合调用toArray()函数转化为数组a（如果传入的集合为null，这一步将抛 出空指针异常），并取得数组的长度numNew；

②modCount自增长1，检查elementData 数组是否需要扩容，需要扩充则扩充；

③然后调用System.arraycopy()方法将数组a中的对象储存到本集合的末尾；

④最后size增加numNew。

### boolean addAll(int index, Collection<? extends E> c)

方法作用：从指定的位置开始，将指定集合中的所有元素插入到此列表中。

①检查插入的索引位置index是否在0到size之间，如果不是，则抛出索引越界异常；

②将传入的集合调用toArray()方法转化为数组a（如果传入的集合为null，这一步将抛 出空指针异常），并取得数组的长度numNew；

③modCount自增长1，检查elementData 数组是否需要扩容，需要扩充则扩充；

④如果index正好等于size，则直接将数组a中的对象依次放到本集合的末尾；否则 先将elementData 数组从index索引到size-1处的对象向后移动numNew距离，再 将 数组a中的对象依次从elementData 数组index索引处开始存放；

⑤最后size增加numNew。

### Void clear()

方法做用：从列表中删除所有元素。

①modCount自增1；

②将elementData 数组中的所有索引出的引用置为null；

③size置为0；

### Object clone()

方法做用：返回此 ArrayList实例的浅拷贝（克隆出的集合中对象引用与原集合中的相 同）。

①调用Object.clone()方法克隆出新的集合v，再调用Arrays.copyOf()方法将原集合中的 对象复制到新集合中，并将新集合的modCount置为0，最后返回集合v。

### boolean contains(Object o)

方法做用：如果此列表包含指定的元素，则返回 true 。

1. 如果o为空，则遍历elementData 数组，依次判断数组中的对象是否为空，若为空， 则直接返回此索引值；如果o不为空，也遍历elementData 数组，调用o.equals()方法 依次与遍历出的对象比较，如果为true，则直接返回此索引值；如果未找到相等的对象， 返回-1；
2. 如果第①步中的返回值>=0，则返回true，否则返回false。

### void ensureCapacity(int minCapacity)

方法做用：如果需要，增加此 ArrayList实例的容量，以确保它可以至少保存minCapacity 个元素。

### E get(int index)

方法做用：返回此列表中指定位置的元素。

### int indexOf(Object o)

方法做用：返回此列表中指定元素的第一次出现的索引，如果此列表不包含元素，则返 回-1。

### Iterator<E> iterator()

方法做用：以正确的顺序返回该列表中的元素的迭代器。

迭代器中的成员变量：

①int cursor：下一个要返回元素的索引值，初始值为0

②int lastRet：最后一个已经返回元素的索引值。初始值为-1

③int expectedModCount：集合被修改的次数，初始值为modCound

迭代器中的方法：

①boolean hasNext()：集合中是否存在下一个元素

当cursor等于size时返回false，否则返回true

②E next()：返回集合中下一个元素

首先检查集合是否被除此迭代器以外的对象做了修改，及此迭代器维护的 expectedModCount是否还等于当前集合的modCound，如果不相等，直接抛出 ConcurrentModificationException异常；

判断cursor是否大于等于size，如果是直接抛出NoSuchElementException异常；

判断cursor是否大于等于elementData.lengh，如果是直接抛出 ConcurrentModificationException异常；

将cursor的值赋值给lastRet，然后cursor自增1；

最后返回elementData[lastRet]。

③void remove()：删除集合中最后被迭代器返回的元素

判断lastRet是否小于0，如果是，直接抛出IllegalStateException异常；

检查集合是否被除此迭代器以外的对象做了修改，及此迭代器维护的 expectedModCount是否还等于当前集合的modCound，如果不相等，直接抛出 ConcurrentModificationException异常；

调用此迭代器所属集合的remove(lastRet)删除集合中最后被迭代器返回的元素（注 意，remove(lastRet)方法中进行了modCound++）；

将lastRet的值赋值给cursor，将lastRet赋值为-1，并将最新的modCound值赋值 给expectedModCount。

④void forEachRemaining(Consumer<? super E> consumer)：对每个剩余元素执行给定的操 作，直到所有元素都被处理或动作引发异常。

判断消费者参数consumer是否为空，若是直接抛出NullPointerException异常；

判断此时迭代器中是否还有剩余的元素，及判断cursor是否大于等于size，是的话 直接返回；

将cursor赋值给i，判断cursor是否大于等于elementData.lengh，如果是直接抛出 ConcurrentModificationException异常；

进入while循环，条件是i != size && modCount == expectedModCount；循环体是 consumer对elementData[i]的操作，之后i++；

将i赋值给cursor，将i-1赋值给lastRet，及lastRet指向集合中最后一个元素的索 引；

最后检查集合是否被除此迭代器以外的对象做了修改，及此迭代器维护的 expectedModCount是否还等于当前集合的modCound，如果不相等，直接抛出 ConcurrentModificationException异常。

### 1.15、int lastIndexOf(Object o)

方法做用：返回此列表中指定元素的最后一次出现的索引，如果此列表不包含元素， 则返回-1。

### 1.16、ListIterator<E> listIterator()

方法做用：返回列表中的列表迭代器（按适当的顺序）。

迭代器中的继承父类的成员变量：

①int cursor：下一个要返回元素的索引值，初始值为0

②int lastRet：最后一个已经返回元素的索引值。初始值为-1

③int expectedModCount：集合被修改的次数，初始值为modCound

迭代器中的方法：

①boolean hasPrevious()：当前迭代器cursor是否指向0索引位置

当cursor等于0时返回false，否则返回true

②int nextIndex()：返回迭代器cursor的值，及下一个元素的索引值

③int previousIndex()：返回迭代器cursor-1的值

④E previous()：返回列表中的上一个元素，并向后移动光标位置。

检查集合是否被除此迭代器以外的对象做了修改，及此迭代器维护的 expectedModCount是否还等于当前集合的modCound，如果不相等，直接抛出 ConcurrentModificationException异常。

判断cursor-1是否小于0，如果是，则直接抛出NoSuchElementException异常；

判断cursor-1是否大于等于elementData.lengh，如果是直接抛出 ConcurrentModificationException异常；

将cursor的值减1，在将cursor的值赋值给lastRet，最后返回elementData[lastRet]。

⑤void set(E e)：用 指定的元素替换由 next()或 previous()返回的最后一个元素。

判断lastRet是否小于0，如果是，则直接抛出IllegalStateException异常；

检查集合是否被除此迭代器以外的对象做了修改，及此迭代器维护的 expectedModCount是否还等于当前集合的modCound，如果不相等，直接抛出 ConcurrentModificationException异常。

调用此迭代器所属的集合的set(lastRet, e)，将传入的e储存到lastRet索引位置。

⑥void add(E e)：将指定的元素插入到下一个元素所在的位置。

检查集合是否被除此迭代器以外的对象做了修改，及此迭代器维护的 expectedModCount是否还等于当前集合的modCound，如果不相等，直接抛出 ConcurrentModificationException异常。

调用此迭代器所属的集合的add(cursor, e)，将传入的e储存到cursor索引位置；

cursor自增1，lastRet赋值为-1，将modCound值赋值给expectedModCount。

### 1.17、ListIterator<E> listIterator(int)

方法做用：从列表中的指定位置开始，返回列表中的元素（按正确顺序）的列表迭代器。

### 1.18、E remove(int index)

方法做用：删除该列表中指定位置的元素。

①判断index >= size，如果是，则抛出IndexOutOfBoundsException异常；

②modCount++；

③调用System.arraycopy()方法将index索引之后的对象往前挪动一个索引的位置；

④将size的值减1，然后将elementData[size]置为空；

⑤最后将elementData[index]处的旧值返回。

### 1.19、boolean remove(Object o)

方法做用：从列表中删除指定元素的第一个出现（如果存在）。

### 1.20、boolean removeAll(Collection<?> c)

方法做用：从此列表中删除指定集合中包含的所有元素。

①判断传入的集合c是否为空。如果是，抛出NullPointerException异常；

②见原码。

### 1.21、void sort(Comparator<? super E> c)

方法做用：使用传入的比较器c对此列表进行重排序。

当集合中元素个数小于32个时，采用插入排序算法；

当集合中元素个数大于31个时，需要阅读java.util.TimSort的原码。

### 1.22、List<E> subList(int fromIndex, int toIndex)

方法做用：返回此列表中指定的 fromIndex （含）和 toIndex之间的视图。

注意：方法返回的List<E>并不是一个新创建的list，而是包含了原集合中fromIndex （含）和 toIndex索引之间的视图，对list的操作实际是对原集合的操作。

### 1.23、Object[] toArray()

方法做用：以正确的顺序（从第一个到最后一个元素）返回一个包含此列表中所有元素的数组。

注意：返回的数组与原集合中的elementData 数组的相对应的每个索引指向同一个引用。

### 1.24、T[] toArray(T[] a)

方法做用：以正确的顺序返回一个包含此列表中所有元素的数组（从第一个到最后一个元素）; 返回的数组的运行时类型是指定数组的运行时类型。

注意：当参数数组a的长度小于集合的长度，返回的数组为新new的数组，其中包含所有集合中的元素；否则，返回的还是数组a本身。

## Vcetor

底层也是数组，无参构造创建的集合数组初始长度为10，当数组装满元素时才会扩容，扩容量为旧数组长度；初始容量和每次的扩容量可以通过带参构造方法进行指定；多数方法都是同步的。

## Stack

继承自Vector，内部只是新增了入栈和出栈方法，实现了栈存储（先进后出）。

## LinkedList

### 构造函数LinkedList()

方法做用：构造一个空列表。

### 构造函数LinkedList(Collection<? extends E> c)

方法做用：按照传入集合c的迭代器返回的顺序，构造一个包含指定集合c所有元素的列表。

调用boolean addAll(Collection<? extends E> c)方法将传入的集合中的元素插入到本集合的链表中。

### boolean add(E e)

方法做用：将指定的元素追加到此列表的末尾。

### void clear()

方法做用：从列表中删除所有元素。

注意：此方法将集合中所有数据节点的值和引用的前后节点全部置为null，让所有节点之间断开关系，方便于垃圾回收。

### Object clone()

方法做用：返回此 LinkedList的浅版本。

源码中为什么要将克隆出来的集合的首尾节点置为空，再将被克隆集合的所有元素添加到克隆集合中，这不是多此一举吗？

### boolean contains(Object o)

方法做用：如果此列表包含指定的元素，则返回 true 。

此方法就是调用集合自身的int indexOf(Object o)方法，查询传入对象在集合中的索引值，只要索引值不等于-1，则返回true。

### Iterator<E> descendingIterator()

方法做用：以逆序返回此集合中的元素的迭代器。

## HashSet

底层基于HashMap，新创建一个hsahset，就是在hsahset内部new一个hashmap，集合中添加的元素作为hashmap的key值，value值统一使用静态最终成员变量PERSENT（一个Object对象）。

## LinkedHashSet

继承HashSet，底层基于LinkedHashMap。

## 5、

## HashMap

### 6.1、构造函数HashMap()

方法作用：将哈希表的加载因子loadFactor赋值为默认值0.75。

### 6.2、构造函数HashMap(int initialCapacity)

方法作用：将哈希表的加载因子loadFactor赋值为默认值0.75，并根据传入的initialCapacity计算出下次数组扩容时，元素数量需要达到的值threshold的值，计算的底层算法使threshold的值只能等于2整数次幂，且最大值为2的30次幂。

### 6.3、构造函数HashMap(int initialCapacity, float loadFactor)

方法作用：将哈希表的加载因子loadFactor赋值为传入的loadFactor，并根据传入的initialCapacity计算出下次数组扩容时，元素数量需要达到的值threshold的值，计算的底层算法使threshold的值只能等于2整数次幂，切最大值为2的30次幂。

### 6.3、构造函数HashMap(Map<? extends K, ? extends V> m)

方法作用：将哈希表的加载因子loadFactor赋值为默认值0.75，其他内容暂留

### 6.4、V put(K key, V value)

方法作用：将指定的 key映射到此 key value中指定的value，及向集合中添加键值对。

①空参构造new出来的实例，默认的数组扩容加载因子是0.75；

②在插入传入的键值对之前，都会对key值计算其hash值；

③如果是首次插入键值对，则需要给集合维护的数组给定初始长度16，数组的扩容阀值为默认加载因子0.75乘以数组长度16等于12；

④在后续插入数据时，根据key的hash值和数组长度减1与运算的结果，决定插入数据存储在数组的哪个索引值下；

⑤如果在需要插入数据的索引值下没有节点数据，则直接将要插入的数据转化为节点后存储在该索引下；

⑥如果该索引下已经存在节点，并且此节点下是单向链表，则按照链表顺序，依次比较已经插入节点的hash值是否相同，如果相同再比较key值（比较地址或者equals比较）是否相同，如果也相同，则将要插入的键值对的valve覆盖此节点的value值；

⑦（接⑥）如果该索引下的所有节点都与要插入的键值对不匹配（匹配方式见第⑥步），则将要插入的键值对转化为节点挂在该索引下单向链表的最末尾；最后判断此单向链表的节点数是否大于8，如果是，则再判断数组table的长度是都是小于64，如果是则对数组进行扩容，若大于等于64，则将此单向链表转化为双向链表，节点之间的顺序不变；接着将此双向链表转化为红黑树（将双向链表首节点作为红黑树初始根节点，双向链表结构保持不变），转化为红黑树之后，如果根节点发生变化，则将新的根节点放到双向链表的首位；

⑧如果该索引下已经存在节点，并且此节点下是红黑树，则从根节点开始比较key值（按第⑥步方式比较），如果有匹配的则将要插入的键值对的value覆盖到匹配的节点的value值；如果没有匹配上任何节点，则按红黑树的规则将要插入的键值对转化为红黑树节点后插入到该红黑树中，并且在双向链表中，要插入节点的父节点作为要插入节点的前接点，要插入节点的父节点的老（旧）后节点作为要插入节点的后节点；如果在红黑树中新增节点后使根节点发生了变化，则需要新的根节点放到双向链表的首位；

⑨如果在判断要插入键值对的hash值等于红黑树中某一节点p的hash值时（hash冲突），判断key是否实现了Comparable接口，如果是，则调用key的compareTo决定大小顺序，如果compareTo返回的结果是0（区分不了大小），则拿到p的左节点p.left，若p.left为空，则p.left等于要插入的键值对，若p.left不为空，则比较p.left与要插入的键值对，知道找到合适的位置；具体过程见原码；

⑩在集合维护的节点数组需要扩容时，当旧数组中某一索引i下存储的是红黑树时，需要对此红黑树进行拆分迁移，从根节点开始，按单向链表遍历整个红黑树，对每个节点进行判断，将hash值和旧数组长度的与运算为0的所有节点以单向链表的形式按遍历顺序连接起来形成单向链表1，其余节点也以单向链表的形式按遍历顺序连接起来形成单向链表2；如果链表1的长度等于0，什么也不做，如果链表1的长度大于0并且小于等于6，则将链表1中的所有红黑树节点转化为单向链表节点，节点之间的单向连接关系不变，红黑树连接关系被抹除，再将链表1的首节点赋值给新数组的i索引下，如果链表1 的长度大于6，先将链表1的首节点赋值给新数组的i索引下，再将链表1转化为红黑树；链表2的步奏和链表2一样，只是在给新数组赋值时，索引值为i加上旧数组的长度；

### 6.5、void clear()

方法作用：将集合维护的数组所有索引的引用都置为null。

注意：此方法只是将集合维护的数组所有索引的引用都置为null，并不处理元素（节点）之间的引用关系。

### 6.6、Object clone()

方法作用：返回此 HashMap实例的浅拷贝：键和值本身不被克隆（注意，克隆出来的集合中维护的table与原集合中的table指向不同的数组对象实例，且数组中储存的节点也是不同的对象实例）。

①调用父类clone方法；

②将上一步的返回值的部分成员变量置为null或者0，其中包括table数组；

③将被克隆集合中的数组中的键值对重新生成节点存储在新克隆出来的数组中。

### 6.7、boolean containsKey(Object key)

方法作用：如果此映射包含指定键的映射，则返回 true 。

①计算出key的hash值，根据hash值确定该在哪个索引下寻找；

②找到查询的索引值后，先判断key是否与首节点的key匹配，是的话则直接返回首节点；

③若key与首节点的key不匹配，则判断首节点的下一个节点是否为空，若为空，直接返回空；

④若首节点的下一个节点不为空，则判断首节点是否为红黑树节点，若是，则从红黑树根节点开始向下匹配，如果有相匹配的节点，则返回此节点，否则返回空；

⑤若首节点不是红黑树节点，则从首节点开始，按单向链表足一匹配，若匹配上，则返回匹配的节点，否则返回空。

### 6.8、Set<Map.Entry<K,V>> entrySet

方法作用：返回此map集合中包含的映射的Set集合，及本hashmap的所有节点的set结合。

## HashTable

与hashmap差异：

同步；

无参构造初始数组容量为11；

确认元素要存储的数组下标，key的hash值与上0x7FFFFFFF之后，在与数组长度取余得到的值即为数组下标；

扩容之后的容量为旧容量乘以2再加1；

扩容后，需要遍历所有元素，重新计算储存的数组下标；

每个数组下标下存储的是单向链表，并且新插入的节点作为头节点。

所存储的元素的key和value都不能为null，若其中存在null值，则在put时会抛出空指针异常。

## LinkedHashMap

继承自hashmap；在hashmap的hash表结构之外，还维护了一个双向链表记录的元素插入顺序。

### 7.1、LinkedHashMap()

方法作用：调用hashmap的无参构造，并将字段accessOrder设置为false

### 7.2、LinkedHashMap(int initialCapacity)

方法作用：调用hashmap指定初始容量构造，并将字段accessOrder设置为false

### 7.3、LinkedHashMap(int initialCapacity, float loadFactor)

方法作用：调用hashmap指定初始容量和加载因子构造，并将字段accessOrder设置为false

## TreeMap

### 8.1、TreeMap()

方法作用：空参构造，将成员吧变量comparator置为null

### 8.2、TreeMap(Comparator<? super K> comparator)

方法作用：带参构造，将成员吧变量comparator指向传入的参数

8.3、

### 8.5、V put(K key, V value)

方法作用：向集合中插入键值对

①判断根节点是否为空；

②若根节点root为空，校验key值是否为空（为空抛出空指针异常），并且在成员变量comparator为空的情况下校验key值的对象类型是否实现了Comparable接口，若未实现，则抛出类型转化异常；根据传入的键值对创建新的节点，将此节点赋值给root；size赋值为1；modCount++；最后直接return null；

③若根节点root不为空，才有以下之后的逻辑；

④如果comparator不为空；从根节点root开始，使用comparator比较传入key与节点键值，如果返回值为零，则使用传入的value覆盖此节点的value值，并直接返回旧的value值，为正，则向树的右边寻找，为负则向树的左边寻找，直到节点为空；

⑤如果comparator为空；先判断传入key值是否为空，若为空，抛出空指针异常；将传入的key转型为Comparable，如果key的类型未实现Comparable，则会抛出类型转换异常；从根节点root开始，使用传入key的compareTo方法比较树中节点的键值，如果返回值为零，则使用传入的value覆盖此节点的value值，并直接返回旧的value值，为正，则向树的右边寻找，为负则向树的左边寻找，直到节点为空；

⑥如果遍历到整棵树的末端都未找到与传入key相同的节点，则根据传入的参数新建节点，并将新节点挂在最后被遍历出来的非空节点之下，比较结果为负，挂在左侧，否则挂在右侧；

⑦将新插入的节点颜色置为红色，按照红黑树的规则，调整树的结构

### 8.6、boolean containsValue(Object value)

方法作用：判断此集合中是否存在传入的值

①先找到红黑树中最靠左的节点，也就是key值最小的节点，以此节点为起始位置，按key值由小到大遍历整棵树，如果找到相同的value，则返回true，否则返回false