Java ??

- HashMap □□□□□□

HashMap ??????

```
package com.util;

import java.io.IOException;
import java.io.InvalidObjectException;
import java.io.Serializable;
import java.util.AbstractCollection;
import java.util.AbstractSet;
import java.util.Collection;
import java.util.ConcurrentModificationException;
import java.util.Iterator;
import java.util.Map;
import java.util.NoSuchElementException;
import java.util.Set;
import java.util.Spliterator;
import java.util.function.BiConsumer;
import java.util.function.BiFunction;
import java.util.function.Consumer;
import java.util.function.Function;
import sun.misc.SharedSecrets;

public class HashMap<K, V> extends AbstractMap<K, V> implements Map<K, V>, Cloneable,
Serializable {

□private static final long serialVersionUID = 362498820763181265L;

□/*
□ *这个映射通常作为一个桶状哈希表，但是 当箱子太大时，它们会被转换成 TreeNodes，每个结构类似于
□* java.util.TreeMap。大多数方法尝试使用正常的箱子，但是 在适用时中继到TreeNode方法(只需通过检查)
□ * instanceof□个节点)。
□*
□ *可以遍历和遍历TreeNodes的Bins 与其他类似，但额外支持更快的查找 当人口过剩时。
□*
□ *然而，由于绝大多数垃圾箱在 正常使用时不超载，检查是否存在 树箱可能在表方法的过程中被延迟。
□*
□ *树箱(即，其元素都是treenode的箱)是主要由hashCode排序，但在领带的情况下，如果两个
```

□* 元素具有相同的"class C implements Comparable<C>"，类型，

□*

□ *则使用它们的compareTo方法进行排序。(我们 通过反射保守检查泛型类型来验证 this——参见methodcomparableClassFor)。

□*

□ *增加的复杂性 的树箱在提供最坏情况O(log n)时是值得的 当键有不同的哈希值或者是 有序的，因此，性能下降优雅

□* hashCode()方法的意外或恶意使用 返回分布不佳的值，以及 许多键共享一个hashCode，

□*

□ *只要它们也是具有可比性。(如果这两项都不适用，我们可能会浪费大约100万美元 在时间和空间上是不采取行动的两倍 预防措施。但唯一已知的案例源于糟糕的使用者

□* 编程实践已经非常缓慢，这使得 差别不大。) ＊ 因为treenode的大小大约是普通节点的两倍，我们 仅当bin包含足够的节点以保证使用时使用它们

□* (见MapUtilConst.MapUtilConst.TREEIFY_THRESHOLD)。当它们变得太小时(由于)

□ *移除或调整大小)它们被转换回普通的垃圾箱。在 使用分布良好的用户hashCodes，树箱是 很少使用。 理想情况下，在随机哈希码下，的频率

□* 箱中的*节点遵循泊松分布(http://en.wikipedia.org/wiki/Poisson_distribution)与a

□ *参数，默认大小调整平均约为0.5 阈值为0.75，虽然由于 调整粒度。忽略方差，即期望值 列表大小k的出现次数为(exp(-0.5) * pow(0.5,

□ * k) / factorial (k))□第一个值是:

□ *

□ *

□ *笔记 拼音

□*

□ * 0: 0.60653066 1: 0.30326533 2: 0.07581633 3: 0.01263606 4: 0.00157952 5:

□ * 0.00015795 6: 0.00001316 7: 0.00000094 8: 0.00000006 more: less than 1 in ten

□ * million

□ *

□ * bin□的根通常是它的第一个节点。然而,有时 (目前仅在Iterator.remove上)，根节点可能在其他地方，但可以

□* 在父链接之后恢复(方法TreeNode.root())。

□*

□ *所有适用的内部方法都接受一个哈希码作为参数 通常由公共方法提供)，允许它们相互调用 无需重新计算用户的哈希值。大多数内部方法也接受一个“tab”。

□* 参数，通常是当前表，但也可能是新表或旧表 当调整大小或转换时。

□*

□ *当bin列表被树形化、分割或未树形化时，我们将它们保持不变 更好地保存相对访问/遍历顺序(即Node.next字段) 局部性，并稍微简化拆分和遍历的处理

□* 调用iterator.remove方法。在插入时使用比较器时，保留一个总数 在重新平衡时进行排序(或尽可能接近)，我们进行比较

□* classes和identityHashCodes是决定因素。

□*

□ *普通模式和树模式之间的使用和转换是复杂的 存在子类LinkedHashMap。下面是钩子方法的定义

□* 在允许LinkedHashMap内部操作的插入、删除和访问时调用 保持独立于这些机制。(这也要求 将map实例传递给一些可能会创建新节点的工具方法。)

□ *

□ *类似于并发编程的基于ssa的编码风格有助于避免别名 所有扭曲的指针操作都会出错。

□*

□ */

□/* ---------------- Static utilities -------------- */

```
/**
 *计算key.hashCode()，并将哈希值的高位按XORs对齐到低位。因为该表使用2的幂次掩码，所以散列值的集合只在
 * 当前掩码上方的比特位始终会发生碰撞。 (已知的例子有 在小表格中保存连续整数的浮点键集合。) 所以我们
 * 应用一个转换，将高比特位的影响向下扩展。有一个比特扩展的速度、效用和质量之间的权衡。因为很多 公共哈希集合已经合理分布(所以不要从中受益
 * 避免传播)，因为我们使用树来处理大量的碰撞 在bin中，我们只需要以可能的最便宜的方式异或移位一些位即可 系统损耗，以及对合并影响的最高位表示
 * 由于表的边界，否则将永远不会用于索引计算。
 * https://blog.csdn.net/liuxingrong666/article/details/103640412
 */
static final int hash(Object key) {
    int h;
    return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
}

/**
 *把不是2的幂的值改成是2的幂 https://blog.csdn.net/ywb201314/article/details/120022308
 */
static final int tableSizeFor(int cap) {
    int n = cap - 1;
    n |= n >>> 1;
    n |= n >>> 2;
    n |= n >>> 4;
    n |= n >>> 8;
    n |= n >>> 16;
    return (n < 0) ? 1 : (n >= MapUtilConst.MAXIMUM_CAPACITY) ? MapUtilConst.MAXIMUM_CAPACITY : n
+ 1;
}

/* ---------------- Fields -------------- */

/**
 * The table, initialized on first use, and resized as necessary. When
 * allocated, length is always a power of two. (We also tolerate length zero in
 * some operations to allow bootstrapping mechanics that are currently not
 * needed.) hash
*/
transient Node<K, V>[] table;

/**
 *保存缓存的entrySet()。注意，AbstractMap字段用于keySet() 和values()。
```

```
 */
transient Set<Map.Entry<K, V>> entrySet;

/**
 * 包含的键值映射的数量。
 */
transient int size;

/**
 * <pre>
 * modCount 是hashmap结构修改的次数。
 * 在之前对iterator（迭代器）进行讲解的时候我已经进行了说明，
 * 需要注意的是在hashmap中modcount指的是结构更改的次数，例如添加新的node，
 * 但是如果是替换原有node的value，modcount是不变的，因为它不属于结构变化。
 * </pre>
 */
transient int modCount;

/**
 * threshold hashmap所能容纳的最大数据量的Node个数，
 * 默认为0.75，threshold=DEFAULT_INITIAL_CAPACITY*loadFactor;当添加元素数量超过这个数量过后，就要进行扩容，扩容后hashmap
的容量是之前的两倍
 */
int threshold;

/**
 * The load factor for the hash table.
 *
 * @serial
 */
final float loadFactor;

/* ---------------- Public operations -------------- */

/**
 * Constructs an empty <tt>HashMap</tt> with the specified initial capacity and
 * load factor.
 *
 * @param initialCapacity the initial capacity
 * @param loadFactor      the load factor
```

```
     * @throws IllegalArgumentException if the initial capacity is negative or the
     *                                  load factor is nonpositive
     */
    public HashMap(int initialCapacity, float loadFactor) {
        if (initialCapacity < 0)
            throw new IllegalArgumentException("Illegal initial capacity: " + initialCapacity);
        if (initialCapacity > MapUtilConst.MAXIMUM_CAPACITY)
            initialCapacity = MapUtilConst.MAXIMUM_CAPACITY;
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new IllegalArgumentException("Illegal load factor: " + loadFactor);
        this.loadFactor = loadFactor;
        this.threshold = tableSizeFor(initialCapacity);
    }

    /**
     * Constructs an empty <tt>HashMap</tt> with the specified initial capacity and
     * the default load factor (0.75).
     *
     * @param initialCapacity the initial capacity.
     * @throws IllegalArgumentException if the initial capacity is negative.
     */
    public HashMap(int initialCapacity) {
        this(initialCapacity, MapUtilConst.DEFAULT_LOAD_FACTOR);
    }

    /**
     * Constructs an empty <tt>HashMap</tt> with the default initial capacity (16)
     * and the default load factor (0.75).
     */
    public HashMap() {
        this.loadFactor = MapUtilConst.DEFAULT_LOAD_FACTOR; // all other fields defaulted
    }

    /**
     * Constructs a new <tt>HashMap</tt> with the same mappings as the specified
     * <tt>Map</tt>. The <tt>HashMap</tt> is created with default load factor (0.75)
     * and an initial capacity sufficient to hold the mappings in the specified
     * <tt>Map</tt>.
     *
     * @param m the map whose mappings are to be placed in this map
```

```
□ * @throws NullPointerException if the specified map is null
□ */
□public HashMap(Map<? extends K, ? extends V> m) {
□□this.loadFactor = MapUtilConst.DEFAULT_LOAD_FACTOR;
□□putMapEntries(m, false);
□}

□/**
□ *□□□□□□□□map□□□□□□HashMap
□ *
□ * <pre>
□
□if (table == null)□□□□□□□HashMap□□□□□□□□□putMapEntries□□□□□□□□□□□□□□□□□□□putAll□
□float ft = ((float)s / loadFactor) + 1.0F
□□□□□1□□□□size / loadFactor = capacity□□□□□□□□capacity□□□□□□□□□□□□□□□□□□□□
□□□□□□□□□capacity□□□□□□□□□□
□
□□□□□□□□□□□□□□MAXIMUM_CAPACITY□□□□□□capacity□□MAXIMUM_CAPACITY□
□
□if (t > threshold)□□□□threshold□□□□□□□□□□capacity□□□□
□□□□table□□□□□□□□table□□null□□□□□□□capacity□□□□□threshold□□□□□□□HashMap□□□□□□□□capacity□capacity
□□□□table□□□□□□□□□□□□□□
□
□else if (s > threshold)□□□□map□size□□□□□□□map□threshold□□□□□map□□□□□□□□map□□□□□□□□□□□□□□□resize
□□□
□□□□□□putVal□□□□□□resize□□
□* </pre>
□ *
□ * Implements Map.putAll and Map constructor.
□ *
□ *
□ * @param m     the map
□ * @param evict false when initially constructing this map, else true (relayed
□ *              to method afterNodeInsertion).
□ */
□final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
□□int s = m.size();

□□if (s <= 0) {
□□□return;
```

```java
        }
        // table table=null putMapEntries

        if (table == null) { // pre-size
            // HashMap Map s HashMap Map
            // HashMap m
            // / = HashMap
            // 2 size 1 ft HashMap
            //
            float ft = ((float) s / loadFactor) + 1.0F;

            // ft t HashMap ft t 1.0f,

            int t = ((ft < (float) MapUtilConst.MAXIMUM_CAPACITY) ? (int) ft :
MapUtilConst.MAXIMUM_CAPACITY);

            // t >
            // ( , HashMap threshold )
            // t , resize() threshold
            if (t > threshold)
                threshold = tableSizeFor(t);
        }
        // Map , map
        // put
        else if (s > threshold)
            resize();
        // map, map key value HashMap
        for (Map.Entry<? extends K, ? extends V> e : m.entrySet()) {
            K key = e.getKey();
            V value = e.getValue();
            putVal(hash(key), key, value, false, evict);
        }
    }

    /**
     * Returns the number of key-value mappings in this map.
     *
     * @return the number of key-value mappings in this map
     */
    public int size() {
```

```
    return size;
}

/**
 * Returns <tt>true</tt> if this map contains no key-value mappings.
 *
 * @return <tt>true</tt> if this map contains no key-value mappings
 */
public boolean isEmpty() {
    return size == 0;
}

/**
 * Returns the value to which the specified key is mapped, or {@code null} if
 * this map contains no mapping for the key.
 *
 * <p>
 * More formally, if this map contains a mapping from a key {@code k} to a value
 * {@code v} such that {@code (key==null ? k==null :
 * key.equals(k))}, then this method returns {@code v}; otherwise it returns
 * {@code null}. (There can be at most one such mapping.)
 *
 * <p>
 * A return value of {@code null} does not <i>necessarily</i> indicate that the
 * map contains no mapping for the key; it's also possible that the map
 * explicitly maps the key to {@code null}. The {@link #containsKey containsKey}
 * operation may be used to distinguish these two cases.
 *
 * @see #put(Object, Object)
 */
public V get(Object key) {
    Node<K, V> e;
    return (e = getNode(hash(key), key)) == null ? null : e.value;
}

/**
 *□□□□□
*
 * <pre>
1.□hash□□□□□
```

2. 判断桶上的key是否等于传入的的key，如果相等，则直接返回。

3. 桶上的key如果不是我们需要的key，则查看next指向的结点。

    3.1如果指向的结点是树结点，调用红黑树的查找方法找到key。

   3.2 如果指向的结点是链表，则遍历链表找到key。

1, map不能为空，且hash对应的下标要存在。否则返回null

2, 取下标对应的对象，如果该对象的key与入参key一致，则返回该对象

3, 否则，获取下一个对象，如果该对象为空，返回null

4, 如果当前对象是树结构，则调用getTreeNode，获取是否存在

5, 如果不是树结构，循环对象，如果存在对象的key与入参key一致，则返回该对象

6, 最终返回要么有值，要么null

```
 *
 *
 * </pre>
 */
final Node<K, V> getNode(int hash, Object key) {
    //要用于将哈希表赋值给tab
    Node<K, V>[] tab;

    //要是通过(哈希表的长度-1) & hash值，计算出数组下标first = tab[n](第一个节点)
    Node<K, V> first,
        主//将当前节点的下个结点赋值给e
        e;

    //要用去哈希表的长度赋值给n
    int n;

    //要用于当前结点的key赋值给k
    K k;

    //哈希表赋值给tab然后判断是否为空
```

```
    // &&hash(length2^16)
    if ((tab = table) == null || (n = tab.length) < 0 || (first = tab[(n - 1) & hash]) == null) {
        return null;
    }

    //(first)hashkeyhaskey
    if (first.hash == hash && // always check first node
        ((k = first.key) == key || (key != null && key.equals(k))))
        return first;

    //firste
    if ((e = first.next) == null) {
        return null;
    }

    //firstTreeNode
    if (first instanceof TreeNode)
        return ((TreeNode<K, V>) first).getTreeNode(hash, key);

    //
    do {
        (e)hashkeyhaskeyfirst(first)
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
            return e;
    } while (null != (e = e.next)); //e
    
    return null;
}

/**
 * Returns <tt>true</tt> if this map contains a mapping for the specified key.
 *
 * @param key The key whose presence in this map is to be tested
 * @return <tt>true</tt> if this map contains a mapping for the specified key.
 */
public boolean containsKey(Object key) {
    return getNode(hash(key), key) != null;
}
```

```
    /**
     * Associates the specified value with the specified key in this map. If the map
     * previously contained a mapping for the key, the old value is replaced.
     *
     * @param key   key with which the specified value is to be associated
     * @param value value to be associated with the specified key
     * @return the previous value associated with <tt>key</tt>, or <tt>null</tt> if
     *         there was no mapping for <tt>key</tt>. (A <tt>null</tt> return can
     *         also indicate that the map previously associated <tt>null</tt> with
     *         <tt>key</tt>.)
     */
    public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }

    /**
     * Implements Map.put and related methods.
     *
     * @param hash         hash for key
     * @param key          the key
     * @param value        the value to put
     * @param onlyIfAbsent 如果为true, 则不更改现有值
     * @param evict        如果为false, 则表处于创建模式.
     * @return previous value, or null if none
     */
    final V putVal(int hash, K key, V value, boolean onlyIfAbsent, boolean evict) {

        //取哈希表中的桶数组和桶数组的长度
        Node<K, V>[] tab;
        Node<K, V> p;
        int n, i;

        /*
          如果table为空, 或者数组当前长度为0, 说明没有创建HashMap, 要先创建
          也说明了HashMap在插入第一个元素的时候才初始化, 而不是定义出来才开始初始化 延迟初始化逻辑
         */
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
```

□□/路由算法（n-1） & hash == hash%n （n必须为2的次方数）

//□□ 如果这个位置为null，则说明其中没有数据，直接放进去

if ((p = tab[i = (n - 1) & hash]) == null)

□□□tab[i] = newNode(hash, key, value, null);

□□/□个位置已经有数据了

else {

□□□□/：临□节点

//□□k：表示临时的一个key

□□□Node<K, V> e;

□□□K k;

□□□□/在上□的if语句中赋值，代表对应数组下标的元素

//□□hash值相同并不一定代表值相同，但如果hash值不相同，值一定不相同

//□□表示桶位中的元素该元素，和你当前插入的key完全一致，表示后续需要进行替换操作

if□□(p.hash == hash &&

□□□□判断基本类型的key值是否相同，直接使用等于判断 || 判断非基本类型的key值是否相同，要使用equals

□□□□□((k = p.key) == key || (key != null && key.equals(k))))

□□□□e = p;

□□□□/值为key值相等的这个元素

//□□key值不相等，且已经发生树化=>红黑树

else if (p instanceof TreeNode)

□□□□e = ((TreeNode<K, V>) p).putTreeVal(this, tab, hash, key, value);

□□□□不是树，且key不相等，剩下一种情况=>链表

//□□链表的头元素和key不一致，则要遍历整个链表，找到key值就替换，不然就在链表末尾插

else {

□□□□for (int binCount = 0;; ++binCount) {

□□□□遍历整个链表

if□□□□e = p.next) == null) {

□□□□一直遍历到空都没有找到key值相等，在表尾插入一个新节点

p.next = newNode(hash, key, value, null);

□□□□□□// MapUtilConst.TREEIFY_THRESHOLD阈值□默认为8

□□□□因为binCount从0开始，所以当binCount为7的时候，已经有八个节点了，binCount在for循环结束的时候才自增，因此要和树化阈值-1作比较

//□□当大于等于阈值的时候，链表树化

if□□□□binCount >= MapUtilConst.TREEIFY_THRESHOLD - 1) // -1 for 1st

□□□□□□□treeifyBin(tab, hash);

```
            // e null
            break;
        }
        // Key
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k))))
            break;

        // e key
        p = e;
    }
}
// if e null
if (e != null) { // existing mapping for key
    // value
    V oldValue = e.value;

    // onlyIfAbsent false key value e.value value
    // onlyIfAbsent true oldvalue if value

    if (!onlyIfAbsent || oldValue == null)
        e.value = value;

    afterNodeAccess(e);

    // oldvalue value
    return oldValue;
  }
}

// modCount
++modCount;

// size
if (++size > threshold)
    resize();
afterNodeInsertion(evict);
return null;
}

/**
```

```java
 * Initializes or doubles table size. If null, allocates in accord with initial
 * capacity target held in field threshold. Otherwise, because we are using
 * power-of-two expansion, the elements from each bin must either stay at same
 * index, or move with a power of two offset in the new table.
 *
 * @return the table
 */
final Node<K, V>[] resize() {

    // [illegible]
    Node<K, V>[] oldTab = table;

    // [illegible]
    int oldCap = (oldTab == null) ? 0 : oldTab.length;

    // [illegible]
    int oldThr = threshold;

    // [illegible]0
    int newCap, newThr = 0;

    // [illegible]0[illegible]
    if (oldCap > 0) {

        // [illegible]2[illegible]30[illegible]
        if (oldCap >= MapUtilConst.MAXIMUM_CAPACITY) {

            // [illegible]int[illegible]2[illegible]31[illegible] -1 [illegible]oldCap[illegible]2[illegible]
            threshold = Integer.MAX_VALUE;

            // [illegible]
            return oldTab;
        }
        /*
         * [illegible]2[illegible]1[illegible]2[illegible] [illegible] [illegible]
         * [illegible]16[illegible]2[illegible]16[illegible]
         * [illegible]16[illegible]
         */
        else if ((newCap = oldCap << 1) < MapUtilConst.MAXIMUM_CAPACITY
                && oldCap >= MapUtilConst.DEFAULT_INITIAL_CAPACITY)
```

```
            newThr = oldThr << 1; // double threshold
        }

        // PS2
        // □□□□else if □□□□□□□□□□
        // □□□□□□□□□□□0□□□□□□□□□□□□□□
        // □□□□□□□□□□map□□□□□□□□□□□□
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        else { // zero initial threshold signifies using defaults

            // □□□□□□□□□□□□□□□□□□□□□□map□□□□□□□□□
            // □□□□□□□□ □ 16
            newCap = MapUtilConst.DEFAULT_INITIAL_CAPACITY;

            // □□□□□□□□ 16*0.75 = 12□0.75□□□□□□□□□□□□□□4□□3□□□□□
            newThr = (int) (MapUtilConst.DEFAULT_LOAD_FACTOR * MapUtilConst.DEFAULT_INITIAL_CAPACITY);
        }
        // □□□□□□0 □PS2□□□□
        if (newThr == 0) {
            float ft = (float) newCap * loadFactor;
            newThr = (newCap < MapUtilConst.MAXIMUM_CAPACITY && ft < (float) MapUtilConst.MAXIMUM_CAPACI
? (int) ft
                : Integer.MAX_VALUE);
        }

        // □□map□□□□□ □□□□
        threshold = newThr;
        @SuppressWarnings({ "rawtypes", "unchecked" })

        // □□□□□□□□□□□□□□□□□□□□□□□□□□□□oldTab□□□□□□□□□□□□□□□□□□□□□
        Node<K, V>[] newTab = (Node<K, V>[]) new Node[newCap];

        // □□map□table□□□□□□□□
        table = newTab;

        // □□□□□□□□□□□□□□□□□□□□□□□□□□□□
        if (oldTab != null) {

            // □□□
```

```
for (int j = 0; j < oldCap; ++j) {
    Node<K, V> e;
    // □□□□□□□□□□□□□□□□□□□□□□□□□□
    if ((e = oldTab[j]) != null) {
        // □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
        oldTab[j] = null;

        // □□□□□□□□□□□□□□□□□□□hash□□
        if (e.next == null)
            // PS3
            // □□□□□□□□□□□□□□□□□□□□□□□□□hash□□□□□□□□□□□□
// □□hash□ % □□□□□ = □ hash□ & □□□□□-1□□
// □□□□□□□□□□□□□□ □□□□□□□2□N□□□□□□□□□□□□□□□□□□□□□□□□□□
// □□□□□□□17,15□□□□□□□□□□□□□□□□□□□□□□□tableSizeFor□□□□□□□□□□□□□□□□□□□□□□2□N□□□
// □□15□-> 16□17-> 32
            newTab[e.hash & (newCap - 1)] = e;

        // □□□□□□□□□□□□□□□□□□□□□□□□□□□□□hash□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
// □□□□□□□□□□16□□□hash□□1□1%16=1□□□hash□□17□17%16=1□□□□□□□□□□□□□□□□□2□□□□□□□□□□□□□1□□□□□□□□32□1%32=1□□□hash□□1□□□□□□□□□□□□□□□□□□□□□□hash□□17□17%32=17□□□□□□□□□□□□□□18□□□□□□
// □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
// □□□□□□□□TreeNode□□
        else if (e instanceof TreeNode)
            ((TreeNode<K, V>) e).split(this, newTab, j, oldCap);
        else { // preserve order

            // □□□□□□□□□□□□□□□
            Node<K, V> loHead = null, loTail = null;
            // □□□□□□□□□□□□□□□
            Node<K, V> hiHead = null, hiTail = null;

            // □□□□□□□ 0 □ oldCap-1 □□□□□□□oldCap □ newCap - 1
            Node<K, V> next;

            //
            do {
                next = e.next;
                // □□□□□hash□ □ □□□□□□ □□□□
// PS3□□□□□□□□□□□□□□□2□N□□□□□16□□□□hash□□□□□□□□□□□□□□0□□□□□hash□□□□□□□□□□□hash□□7□□
// □□□□hash□□□□□□□□□□□□□□mod□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

```
            if ((e.hash & oldCap) == 0) {
                // PS4
                    // □□□□□
                if (loTail == null)
                    // □□□□□□
                    loHead = e;
                else
                    // □□□□□□□□□□□□□□□□□□□
                    loTail.next = e;

                    // □□□□
                loTail = e;
            }

            // □□□0□□□hash□□□□□□□□□□□hash□□17□
// □□□□□□□□□□□□□□□□□□□□□
// □□□□□□□□□16□□□□□□□□□32□hash□17□□□□□□□□□□17□□□□□□□□□□□16□□□□□□16□□□□□□□□□□□□[0-15]□□□□[16-31]

            else {

                // PS4
                if (hiTail == null)
                    hiHead = e;
                else
                    hiTail.next = e;
                hiTail = e;
            }
        } while ((e = next) != null);

        // □□□□□□□□□□□□□□□□
        if (loTail != null) {
            loTail.next = null;
            newTab[j] = loHead;
        }

        if (hiTail != null) {
            // □□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
            hiTail.next = null;

            // hash□ 17 □□□□□□□0□□□□□□□□□□16□□□ hash□ 18 □□□□□□□1□□□□□□□□□□17□□□
```

```
                    newTab[j + oldCap] = hiHead;
                }
              }
            }
          }
        }
        return newTab;
    }

    /**
     * <pre>
     * 1□□table□□□□□□□□□□□□64□□□□table□□
     * 2□□table□□□□64□□□□□□□□□□□□
     * 2.1□□□□□□□□□□□□□□□□□□□□□□□□□□□
     * 2.2□□□□□□□□□□□
     * </pre>
     */
    final void treeifyBin(Node<K, V>[] tab, int hash) {
        int n, index;
        Node<K, V> e;

        // □1□table□□□□□□□□□□□□64□□□□table□□
        if (tab == null || (n = tab.length) < MapUtilConst.MIN_TREEIFY_CAPACITY)
            resize();

        // □2□table□□□□64□□□□□□□□□□□□
        else if ((e = tab[index = (n - 1) & hash]) != null) {
            TreeNode<K, V> hd = null, tl = null;
            do {
                // □2.1□□□□□□□□□□□□□□□□□□□□□□□□□
                TreeNode<K, V> p = replacementTreeNode(e, null);
                if (tl == null)
                    hd = p;
                else {
                    // □□□□
                    p.prev = tl;
                    tl.next = p;
                }
                tl = p;
            } while ((e = e.next) != null);
```

```
            if ((tab[index] = hd) != null)
                hd.treeify(tab);
        }
    }

    /**
     * Copies all of the mappings from the specified map to this map. These mappings
     * will replace any mappings that this map had for any of the keys currently in
     * the specified map.
     *
     * @param m mappings to be stored in this map
     * @throws NullPointerException if the specified map is null
     */
    public void putAll(Map<? extends K, ? extends V> m) {
        putMapEntries(m, true);
    }

    /**
     * Removes the mapping for the specified key from this map if present.
     *
     * @param key key whose mapping is to be removed from the map
     * @return the previous value associated with <tt>key</tt>, or <tt>null</tt> if
     *         there was no mapping for <tt>key</tt>. (A <tt>null</tt> return can
     *         also indicate that the map previously associated <tt>null</tt> with
     *         <tt>key</tt>.)
     */
    public V remove(Object key) {
        Node<K, V> e;
        return (e = removeNode(hash(key), key, null, false, true)) == null ? null : e.value;
    }

    /**
     * Implements Map.remove and related methods.
     *
     * @param hash       hash for key
     * @param key        the key
     * @param value      the value to match if matchValue, else ignored
     * @param matchValue if true only remove if value is equal
     * @param movable    if false do not move other nodes while removing
     * @return the node, or null if none
```

```
 */
final Node<K, V> removeNode(int hash, Object key, Object value, boolean matchValue, boolean
movable) {

    // tab
    //p
    Node<K, V>[] tab;
    Node<K, V> p;
    int n, index;

    //
    //  0
    // hash                p  null
    if ((tab = table) != null && (n = tab.length) > 0 && (p = tab[index = (n - 1) & hash]) !=
null) {

        // node                e                /
        //k e key v                node  value
        Node<K, V> node = null, e;
        K k;
        V v;

        //                -
        //                p                node                p
        if (p.hash == hash && ((k = p.key) == key || (key != null && key.equals(k))))
            node = p;

        //                                                
        //
        else if ((e = p.next) != null) {
            //
            if (p instanceof TreeNode)

                //                                getTreeNode()
                node = ((TreeNode<K, V>) p).getTreeNode(hash, key);

            //                                                
            else {
```

```
            //□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□
    do {
        e//□□□□□□□□□□□□□□node□□e□□□□□□
        if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
            node = e;
            break;
        }
        //□e□□□□□□p□□□e□□□
//□□□□while□□e□□□□□□□□□□□□e□□□□□□□□□□□□□□□□□□□p□□□e□□□□□□□□□
//□□□□□□□□□□□□□□□□□□p□□□□e□□□□□□□□□
        p = e;
    } while ((e = e.next) != null);//□□e□□□□□□□□□□□null□□□□□□□□null□□□□□□□□□□
}
}

    //□□□□ matchValue□true□□□□□□□□□□□□□□false□□□□□□□□□□□□□key□hash□□□□□□□□□□□
    if (node != null && (!matchValue || (v = node.value) == value || (value != null &&
value.equals(v)))) {
        //□□□□□□□□
        if (node instanceof TreeNode)
            //□□□□□□□□
            ((TreeNode<K, V>) node).removeTreeNode(this, tab, movable);

        //□□p□□□
//□□□□□□□ □□□□□□□p□□□□□□□□□□□□□□□□□□□node□□□□□□□p
        else if (node == p)
            //□□□□□□□□□□□□□□□□□□□□□□□□□□
//□□□□□□□□□□□□□□□□□□□□□□□□□
            tab[index] = node.next;

        //□□□□□□□□□□□□□□□
        else
            //□p□node□□□□□□□p.next□□□□□□□□node
            //p.next □□□□□□node□next□□□□□□□□□□node□□□node□□□□
            p.next = node.next;

        //map□□□□□□□□□□□□□□□□□
        ++modCount;

        //table□□□□□□□□□□□□□□□□□□□□□□□□□□□
```

--size;

该在hashMap中是空方法，主要是供LinkedHashMap使用，因为LinkedHashMap重写了该方法

afterNodeRemoval(node);

返回移除的节点

return node;

}

}

//找到 返回null

return null;

}

/**

 * Removes all of the mappings from this map. The map will be empty after this

 * call returns.

 */

public void clear() {

Node<K, V>[] tab;

modCount++;

if ((tab = table) != null && size > 0) {

size = 0;

for (int i = 0; i < tab.length; ++i)

tab[i] = null;

}

}

/**

 * Returns <tt>true</tt> if this map maps one or more keys to the specified

 * value.

 *

 * @param value value whose presence in this map is to be tested

 * @return <tt>true</tt> if this map maps one or more keys to the specified

 *   value

 */

public boolean containsValue(Object value) {

Node<K, V>[] tab;

V v;

if ((tab = table) != null && size > 0) {

for (int i = 0; i < tab.length; ++i) {

```
        for (Node<K, V> e = tab[i]; e != null; e = e.next) {
            if ((v = e.value) == value || (value != null && value.equals(v)))
                return true;
        }
    }
    return false;
}

/**
 * Returns a {@link Set} view of the keys contained in this map. The set is
 * backed by the map, so changes to the map are reflected in the set, and
 * vice-versa. If the map is modified while an iteration over the set is in
 * progress (except through the iterator's own <tt>remove</tt> operation), the
 * results of the iteration are undefined. The set supports element removal,
 * which removes the corresponding mapping from the map, via the
 * <tt>Iterator.remove</tt>, <tt>Set.remove</tt>, <tt>removeAll</tt>,
 * <tt>retainAll</tt>, and <tt>clear</tt> operations. It does not support the
 * <tt>add</tt> or <tt>addAll</tt> operations.
 *
 * @return a set view of the keys contained in this map
 */
public Set<K> keySet() {
    Set<K> ks = keySet;
    if (ks == null) {
        ks = new KeySet();
        keySet = ks;
    }
    return ks;
}

final class KeySet extends AbstractSet<K> {
    public final int size() {
        return size;
    }

    public final void clear() {
        HashMap.this.clear();
    }
```

```
    public final Iterator<K> iterator() {
        return new KeyIterator();
    }

    public final boolean contains(Object o) {
        return containsKey(o);
    }

    public final boolean remove(Object key) {
        return removeNode(hash(key), key, null, false, true) != null;
    }

    public final Spliterator<K> spliterator() {
        return new KeySpliterator<>(HashMap.this, 0, -1, 0, 0);
    }

    public final void forEach(Consumer<? super K> action) {
        Node<K, V>[] tab;
        if (action == null)
            throw new NullPointerException();
        if (size > 0 && (tab = table) != null) {
            int mc = modCount;
            for (int i = 0; i < tab.length; ++i) {
                for (Node<K, V> e = tab[i]; e != null; e = e.next)
                    action.accept(e.key);
            }
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
    }
}

/**
 * Returns a {@link Collection} view of the values contained in this map. The
 * collection is backed by the map, so changes to the map are reflected in the
 * collection, and vice-versa. If the map is modified while an iteration over
 * the collection is in progress (except through the iterator's own
 * <tt>remove</tt> operation), the results of the iteration are undefined. The
 * collection supports element removal, which removes the corresponding mapping
 * from the map, via the <tt>Iterator.remove</tt>, <tt>Collection.remove</tt>,
```

```
     * <tt>removeAll</tt>, <tt>retainAll</tt> and <tt>clear</tt> operations. It does
     * not support the <tt>add</tt> or <tt>addAll</tt> operations.
     *
     * @return a view of the values contained in this map
     */
    public Collection<V> values() {
        Collection<V> vs = values;
        if (vs == null) {
            vs = new Values();
            values = vs;
        }
        return vs;
    }

    final class Values extends AbstractCollection<V> {
        public final int size() {
            return size;
        }

        public final void clear() {
            HashMap.this.clear();
        }

        public final Iterator<V> iterator() {
            return new ValueIterator();
        }

        public final boolean contains(Object o) {
            return containsValue(o);
        }

        public final Spliterator<V> spliterator() {
            return new ValueSpliterator<>(HashMap.this, 0, -1, 0, 0);
        }

        public final void forEach(Consumer<? super V> action) {
            Node<K, V>[] tab;
            if (action == null)
                throw new NullPointerException();
            if (size > 0 && (tab = table) != null) {
```

```
        int mc = modCount;
        for (int i = 0; i < tab.length; ++i) {
            for (Node<K, V> e = tab[i]; e != null; e = e.next)
                action.accept(e.value);
        }
        if (modCount != mc)
            throw new ConcurrentModificationException();
    }
}

/**
 * Returns a {@link Set} view of the mappings contained in this map. The set is
 * backed by the map, so changes to the map are reflected in the set, and
 * vice-versa. If the map is modified while an iteration over the set is in
 * progress (except through the iterator's own <tt>remove</tt> operation, or
 * through the <tt>setValue</tt> operation on a map entry returned by the
 * iterator) the results of the iteration are undefined. The set supports
 * element removal, which removes the corresponding mapping from the map, via
 * the <tt>Iterator.remove</tt>, <tt>Set.remove</tt>, <tt>removeAll</tt>,
 * <tt>retainAll</tt> and <tt>clear</tt> operations. It does not support the
 * <tt>add</tt> or <tt>addAll</tt> operations.
 *
 * @return a set view of the mappings contained in this map
 */
public Set<Map.Entry<K, V>> entrySet() {
    Set<Map.Entry<K, V>> es;
    return (es = entrySet) == null ? (entrySet = new EntrySet()) : es;
}

final class EntrySet extends AbstractSet<Map.Entry<K, V>> {
    public final int size() {
        return size;
    }

    public final void clear() {
        HashMap.this.clear();
    }

    public final Iterator<Map.Entry<K, V>> iterator() {
```

```
        return new EntryIterator();
    }

    public final boolean contains(Object o) {
        if (!(o instanceof Map.Entry))
            return false;
        Map.Entry<?, ?> e = (Map.Entry<?, ?>) o;
        Object key = e.getKey();
        Node<K, V> candidate = getNode(hash(key), key);
        return candidate != null && candidate.equals(e);
    }

    public final boolean remove(Object o) {
        if (o instanceof Map.Entry) {
            Map.Entry<?, ?> e = (Map.Entry<?, ?>) o;
            Object key = e.getKey();
            Object value = e.getValue();
            return removeNode(hash(key), key, value, true, true) != null;
        }
        return false;
    }

    public final Spliterator<Map.Entry<K, V>> spliterator() {
        return new EntrySpliterator<>(HashMap.this, 0, -1, 0, 0);
    }

    public final void forEach(Consumer<? super Map.Entry<K, V>> action) {
        Node<K, V>[] tab;
        if (action == null)
            throw new NullPointerException();
        if (size > 0 && (tab = table) != null) {
            int mc = modCount;
            for (int i = 0; i < tab.length; ++i) {
                for (Node<K, V> e = tab[i]; e != null; e = e.next)
                    action.accept(e);
            }
            if (modCount != mc)
                throw new ConcurrentModificationException();
        }
    }
```

```
    }

    // Overrides of JDK8 Map extension methods

    @Override
    public V getOrDefault(Object key, V defaultValue) {
        Node<K, V> e;
        return (e = getNode(hash(key), key)) == null ? defaultValue : e.value;
    }

    @Override
    public V putIfAbsent(K key, V value) {
        return putVal(hash(key), key, value, true, true);
    }

    @Override
    public boolean remove(Object key, Object value) {
        return removeNode(hash(key), key, value, true, true) != null;
    }

    @Override
    public boolean replace(K key, V oldValue, V newValue) {
        Node<K, V> e;
        V v;
        if ((e = getNode(hash(key), key)) != null && ((v = e.value) == oldValue || (v != null &&
v.equals(oldValue)))) {
            e.value = newValue;
            afterNodeAccess(e);
            return true;
        }
        return false;
    }

    @Override
    public V replace(K key, V value) {
        Node<K, V> e;
        if ((e = getNode(hash(key), key)) != null) {
            V oldValue = e.value;
            e.value = value;
            afterNodeAccess(e);
```

```
□□□return oldValue;
□□}
□□return null;
□}

□/*
□ *□□□□□□□□key□hashMap□□□□□□□□key□□□□□□□□key□□□value
□ *□□key□□□□□□□□□□□□□(hash,key,value,null),value□□□□□ V v =
□ * mappingFunction.apply(key);□□□□□□□□□□□mappingFunction.apply(key)□□□□□
□*/
□@Override
□public V computeIfAbsent(K key, Function<? super K, ? extends V> mappingFunction) {

□□//□□□□□□□□□□□□□□□□□□□
□□if (mappingFunction == null)
□□□throw new NullPointerException();

□□//□□□□key□hash□
□□int hash = hash(key);

□□//□□node□□□□□tab,node□□□□□first,□□n,i;
□□Node<K, V>[] tab;
□□Node<K, V> first;
□□int n, i;

□□//□□□□□□□□□□
□□int binCount = 0;

□□//□□TreeNode□□□□□t;
□□TreeNode<K, V> t = null;

□□//□□□□node□□□□□old□
□□Node<K, V> old = null;
□□if (size > threshold || (tab = table) == null || (n = tab.length) == 0)
□□□//□hashMap□□□□□□□□□□□□table□tab□□□□□□
//□□□□ resize()□□□□□□□□hashMap,□□□□hashMap□□□□
n□□= (tab = resize()).length;
□□if ((first = tab[i = (n - 1) & hash]) != null) {

□□□//□key□tab□□□□□□□□□□□□□□□□□□□□□□□□□□□□□first
```

```
□□□//□□key□□□□□key□□value□oldValue□key□□□□□□
//□□int mc = modCount;□□□□□□□□□□□node□□□key ,value□□□□□first□null,

□□□if (first instanceof TreeNode)

□□□□//□□□□□□□□□□key□□□□old,t;
□□□□old = (t = (TreeNode<K, V>) first).getTreeNode(hash, key);
□□□else {
□□□□//□□□□□□□□□□□□□□□□
//□□□first□□□□□e,□□k;
□□□□Node<K, V> e = first;
□□□□K k;
□□□□do {
□□□□□do-while□□□□□□□□□□□□□hash□□key□□□□□□□□□□□□□□□□□□e□□□old,□□□□□□;
□□□□□if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
□□□□□□old = e;
□□□□□□break;
□□□□□}
□□□□□□//□□□□+1□
++binCount;
□□□□} while ((e = e.next) != null);□//□□□□□□□□□
}□□
□□□V oldValue;
□□□if (old != null && (oldValue = old.value) != null) {
□□□□//□□□□□□□old□□□□□□old□□□value□□□□□□old□value
□□□□/□□□□□
afterNodeAccess(old);
□□□□return oldValue;
□□□}
□□}

□□/**
□□ * mappingFunction.apply(key)□□□□□computeIfAbsent□□□□□□apply□□□□□□key
□□ * apply(key)□□□□□□□□□□□□□□apply□□□□□ R apply(T t); □□□□R□□□Function<T, R>□□□□□□□
□□ * □□□□□□□□□□□□□□□□□□□□ Integer integer1 = map.computeIfAbsent("4", (s)->new
□□ * Integer(6));□□computeIfAbsent□□□□□:computeIfAbsent(K key,Function<? super K,
□□ * ? extends V> mappingFunction)□□□key□4□s=key□□□4□Function<? super 4, ?
□□ * extends V>□□new Integer(6)□□□□□□□□V□□□□6□ □□Function<T, R> --->Function<4,
□□ * 6>---> R apply(T 4),R=6;□□mappingFunction.apply(key)□□□6□□□v=6;
□□ */
```

```
□□V v = mappingFunction.apply(key);
□□if (v == null) {
□□□//□□□□□mappingFunction□□□value□□□□□□null;
□□□return null;
□□} else if (old != null) {
□□□//old□□□□□v□□□old□value
□□□old.value = v;
□□□//□□
afterNodeAccess(old);
□□□return v;
□□}
□□//□□□□□□□□□t□□□□□
else if (t != null)
□□□//□map□□□tab,hash□key□v□□□□□□
t.putTreeVal(this, tab, hash, key, v);
□□else {
□□□//□tab□□□□□□□□(hash, key, v, first),□□first□null,□□first = tab[i = (n - 1) &
□□□// hash]) = null

□□□tab[i] = newNode(hash, key, v, first);
□□□//□□□□□□□□□□□□□□□□
if (binCount >= MapUtilConst.TREEIFY_THRESHOLD - 1)
□□□//tab□□□□□□
treeifyBin(tab, hash);
□□}
□□//□hashMap□□□□□
++modCount;
□□// hashMap□□□1
□□++size;
□□//□□□
afterNodeInsertion(true);
□□return v;
□}

□/**
□ *□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□□ Map □□
□*/
□public V computeIfPresent(K key, BiFunction<? super K, ? super V, ? extends V>
remappingFunction) {
□□if (remappingFunction == null)
```

```
□□□throw new NullPointerException();
□□Node<K, V> e;
□□V oldValue;
□□int hash = hash(key);
□□if ((e = getNode(hash, key)) != null && (oldValue = e.value) != null) {
□□□V v = remappingFunction.apply(key, oldValue);
□□□if (v != null) {
□□□□e.value = v;
□□□□afterNodeAccess(e);
□□□□return v;
□□□} else
□□□□removeNode(hash, key, null, false, true);
□□}
□□return null;
□}

□/**
□ *□□□□□□□□□ key □□□□□□ key □ remappingFunction □□□□□□□□□□□□□□□□□□□□□□□□
□* □□□□□□□□□□ key □□□□□□□ remappingFunction □□□□□□□□□□□□□□□□ key □□□□□
□*/
□@Override
□public V compute(K key, BiFunction<? super K, ? super V, ? extends V> remappingFunction) {
□□if (remappingFunction == null)
□□□throw new NullPointerException();
□□int hash = hash(key);
□□Node<K, V>[] tab;
□□Node<K, V> first;
□□int n, i;
□□int binCount = 0;
□□TreeNode<K, V> t = null;
□□Node<K, V> old = null;
□□if (size > threshold || (tab = table) == null || (n = tab.length) == 0)
□□□n = (tab = resize()).length;
□□if ((first = tab[i = (n - 1) & hash]) != null) {
□□□if (first instanceof TreeNode)
□□□□old = (t = (TreeNode<K, V>) first).getTreeNode(hash, key);
□□□else {
□□□□Node<K, V> e = first;
□□□□K k;
□□□□do {
```

```
                if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
                    old = e;
                    break;
                }
                ++binCount;
            } while ((e = e.next) != null);
        }
    }
    V oldValue = (old == null) ? null : old.value;
    V v = remappingFunction.apply(key, oldValue);
    if (old != null) {
        if (v != null) {
            old.value = v;
            afterNodeAccess(old);
        } else
            removeNode(hash, key, null, false, true);
    } else if (v != null) {
        if (t != null)
            t.putTreeVal(this, tab, hash, key, v);
        else {
            tab[i] = newNode(hash, key, v, first);
            if (binCount >= MapUtilConst.TREEIFY_THRESHOLD - 1)
                treeifyBin(tab, hash);
        }
        ++modCount;
        ++size;
        afterNodeInsertion(true);
    }
    return v;
}

@Override
public V merge(K key, V value, BiFunction<? super V, ? super V, ? extends V>
remappingFunction) {
    if (value == null)
        throw new NullPointerException();
    if (remappingFunction == null)
        throw new NullPointerException();
    int hash = hash(key);
    Node<K, V>[] tab;
```

```
    Node<K, V> first;
    int n, i;
    int binCount = 0;
    TreeNode<K, V> t = null;
    Node<K, V> old = null;
    if (size > threshold || (tab = table) == null || (n = tab.length) == 0)
        n = (tab = resize()).length;
    if ((first = tab[i = (n - 1) & hash]) != null) {
        if (first instanceof TreeNode)
            old = (t = (TreeNode<K, V>) first).getTreeNode(hash, key);
        else {
            Node<K, V> e = first;
            K k;
            do {
                if (e.hash == hash && ((k = e.key) == key || (key != null && key.equals(k)))) {
                    old = e;
                    break;
                }
                ++binCount;
            } while ((e = e.next) != null);
        }
    }
    if (old != null) {
        V v;
        if (old.value != null)
            v = remappingFunction.apply(old.value, value);
        else
            v = value;
        if (v != null) {
            old.value = v;
            afterNodeAccess(old);
        } else
            removeNode(hash, key, null, false, true);
        return v;
    }
    if (value != null) {
        if (t != null)
            t.putTreeVal(this, tab, hash, key, value);
        else {
            tab[i] = newNode(hash, key, value, first);
```

```
                if (binCount >= MapUtilConst.TREEIFY_THRESHOLD - 1)
                    treeifyBin(tab, hash);
                }
                ++modCount;
                ++size;
                afterNodeInsertion(true);
            }
            return value;
        }

        @Override
        public void forEach(BiConsumer<? super K, ? super V> action) {
            Node<K, V>[] tab;
            if (action == null)
                throw new NullPointerException();
            if (size > 0 && (tab = table) != null) {
                int mc = modCount;
                for (int i = 0; i < tab.length; ++i) {
                    for (Node<K, V> e = tab[i]; e != null; e = e.next)
                        action.accept(e.key, e.value);
                }
                if (modCount != mc)
                    throw new ConcurrentModificationException();
            }
        }

        @Override
        public void replaceAll(BiFunction<? super K, ? super V, ? extends V> function) {
            Node<K, V>[] tab;
            if (function == null)
                throw new NullPointerException();
            if (size > 0 && (tab = table) != null) {
                int mc = modCount;
                for (int i = 0; i < tab.length; ++i) {
                    for (Node<K, V> e = tab[i]; e != null; e = e.next) {
                        e.value = function.apply(e.key, e.value);
                    }
                }
                if (modCount != mc)
                    throw new ConcurrentModificationException();
```

```
    }
}

/* ------------------------------------------------------------ */
// Cloning and serialization

/**
 * Returns a shallow copy of this <tt>HashMap</tt> instance: the keys and values
 * themselves are not cloned.
 *
 * @return a shallow copy of this map
 */
@SuppressWarnings("unchecked")
@Override
public Object clone() {
    HashMap<K, V> result;
    try {
        result = (HashMap<K, V>) super.clone();
    } catch (CloneNotSupportedException e) {
        // this shouldn't happen, since we are Cloneable
        throw new InternalError(e);
    }
    result.reinitialize();
    result.putMapEntries(this, false);
    return result;
}

// These methods are also used when serializing HashSets
final float loadFactor() {
    return loadFactor;
}

final int capacity() {
    return (table != null) ? table.length : (threshold > 0) ? threshold :
MapUtilConst.DEFAULT_INITIAL_CAPACITY;
}

/**
 * Save the state of the <tt>HashMap</tt> instance to a stream (i.e., serialize
 * it).
```

```
    *
    * @serialData The <i>capacity</i> of the HashMap (the length of the bucket
    *             array) is emitted (int), followed by the <i>size</i> (an int, the
    *             number of key-value mappings), followed by the key (Object) and
    *             value (Object) for each key-value mapping. The key-value mappings
    *             are emitted in no particular order.
    */
    private void writeObject(java.io.ObjectOutputStream s) throws IOException {
        int buckets = capacity();
        // Write out the threshold, loadfactor, and any hidden stuff
        s.defaultWriteObject();
        s.writeInt(buckets);
        s.writeInt(size);
        internalWriteEntries(s);
    }

    /**
    * Reconstitutes this map from a stream (that is, deserializes it).
    *
    * @param s the stream
    * @throws ClassNotFoundException if the class of a serialized object could not
    *                                be found
    * @throws IOException            if an I/O error occurs
    */
    private void readObject(java.io.ObjectInputStream s) throws IOException,
    ClassNotFoundException {
        // Read in the threshold (ignored), loadfactor, and any hidden stuff
        s.defaultReadObject();
        reinitialize();
        if (loadFactor <= 0 || Float.isNaN(loadFactor))
            throw new InvalidObjectException("Illegal load factor: " + loadFactor);
        s.readInt(); // Read and ignore number of buckets
        int mappings = s.readInt(); // Read number of mappings (size)
        if (mappings < 0)
            throw new InvalidObjectException("Illegal mappings count: " + mappings);
        else if (mappings > 0) { // (if zero, use defaults)
            // Size the table using given load factor only if within
            // range of 0.25...4.0
            float lf = Math.min(Math.max(0.25f, loadFactor), 4.0f);
            float fc = (float) mappings / lf + 1.0f;
```

```
        int cap = ((fc < MapUtilConst.DEFAULT_INITIAL_CAPACITY) ?
MapUtilConst.DEFAULT_INITIAL_CAPACITY
                   : (fc >= MapUtilConst.MAXIMUM_CAPACITY) ? MapUtilConst.MAXIMUM_CAPACITY : tableSizeFor((in
fc));
        float ft = (float) cap * lf;
        threshold = ((cap < MapUtilConst.MAXIMUM_CAPACITY && ft < MapUtilConst.MAXIMUM_CAPACITY) ?
(int) ft
                     : Integer.MAX_VALUE);

        // Check Map.Entry[].class since it's the nearest public type to
        // what we're actually creating.
        SharedSecrets.getJavaOISAccess().checkArray(s, Map.Entry[].class, cap);
        @SuppressWarnings({ "rawtypes", "unchecked" })
        Node<K, V>[] tab = (Node<K, V>[]) new Node[cap];
        table = tab;

        // Read the keys and values, and put the mappings in the HashMap
        for (int i = 0; i < mappings; i++) {
            @SuppressWarnings("unchecked")
            K key = (K) s.readObject();
            @SuppressWarnings("unchecked")
            V value = (V) s.readObject();
            putVal(hash(key), key, value, false, false);
        }
    }
}

/* ------------------------------------------------------------ */
// iterators

abstract class HashIterator {
    Node<K, V> next; // next entry to return
    Node<K, V> current; // current entry
    int expectedModCount; // for fast-fail
    int index; // current slot

    HashIterator() {
        expectedModCount = modCount;
        Node<K, V>[] t = table;
        current = next = null;
```

```
        index = 0;
        if (t != null && size > 0) { // advance to first entry
            do {
            } while (index < t.length && (next = t[index++]) == null);
        }
    }

    public final boolean hasNext() {
        return next != null;
    }

    final Node<K, V> nextNode() {
        Node<K, V>[] t;
        Node<K, V> e = next;
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        if (e == null)
            throw new NoSuchElementException();
        if ((next = (current = e).next) == null && (t = table) != null) {
            do {
            } while (index < t.length && (next = t[index++]) == null);
        }
        return e;
    }

    public final void remove() {
        Node<K, V> p = current;
        if (p == null)
            throw new IllegalStateException();
        if (modCount != expectedModCount)
            throw new ConcurrentModificationException();
        current = null;
        K key = p.key;
        removeNode(hash(key), key, null, false, false);
        expectedModCount = modCount;
    }
}

final class KeyIterator extends HashIterator implements Iterator<K> {
    public final K next() {
```

```
        return nextNode().key;
    }
}

final class ValueIterator extends HashIterator implements Iterator<V> {
    public final V next() {
        return nextNode().value;
    }
}

final class EntryIterator extends HashIterator implements Iterator<Map.Entry<K, V>> {
    public final Map.Entry<K, V> next() {
        return nextNode();
    }
}

/* ------------------------------------------------------------ */
// spliterators
/* ------------------------------------------------------------ */
// LinkedHashMap support

/*
 * The following package-protected methods are designed to be overridden by
 * LinkedHashMap, but not by any other subclass. Nearly all other internal
 * methods are also package-protected but are declared final, so can be used by
 * LinkedHashMap, view classes, and HashSet.
 */

// Create a regular (non-tree) node
Node<K, V> newNode(int hash, K key, V value, Node<K, V> next) {
    return new Node<>(hash, key, value, next);
}

// For conversion from TreeNodes to plain nodes
Node<K, V> replacementNode(Node<K, V> p, Node<K, V> next) {
    return new Node<>(p.hash, p.key, p.value, next);
}

// Create a tree bin node
TreeNode<K, V> newTreeNode(int hash, K key, V value, Node<K, V> next) {
```

```
    return new TreeNode<>(hash, key, value, next);
}

// For treeifyBin
TreeNode<K, V> replacementTreeNode(Node<K, V> p, Node<K, V> next) {
    return new TreeNode<>(p.hash, p.key, p.value, next);
}

/**
 * Reset to initial default state. Called by clone and readObject.
 */
void reinitialize() {
    table = null;
    entrySet = null;
    keySet = null;
    values = null;
    modCount = 0;
    threshold = 0;
    size = 0;
}

// Callbacks to allow LinkedHashMap post-actions
void afterNodeAccess(Node<K, V> p) {
}

void afterNodeInsertion(boolean evict) {
}

void afterNodeRemoval(Node<K, V> p) {
}

// Called only from writeObject, to ensure compatible ordering.
void internalWriteEntries(java.io.ObjectOutputStream s) throws IOException {
    Node<K, V>[] tab;
    if (size <= 0 || (tab = table) == null) {
        return;
    }

        for (int i = 0; i < tab.length; ++i) {
            for (Node<K, V> e = tab[i]; e != null; e = e.next) {
```

```
                s.writeObject(e.key);
                s.writeObject(e.value);
            }
□□}
□}
}
```