Collection Studying 4 -- HashMap

1）hashmap底层存储原理

Node数组+链表+红黑树

2）冲突是如何解决的 解决哈希冲突的几种方法

a.拉链法，在每个数组元素拉一个链表，处理冲突的元素

b. rehash(重哈希法)，公共溢出区，开放地址法(hash位置+x判断是否冲突)

3）为何HashMap的数组长度一定是2的次幂

作者逻辑设定的，初始是找到最接近的二次幂容量，扩容翻倍

1）二次幂的长度便于将哈希取模运算变成与运算，以提高速度

2）便于扩容时，快速重哈希，只需将旧桶元素重新分到两个桶

4）hashtable和hashmap区别

1）HashMap是非线程安全的，HashTable是线程安全的；HashTable的方法经过了synchronized 修饰；

2）HashMap效率高，HashMap因为保证线程安全效率低

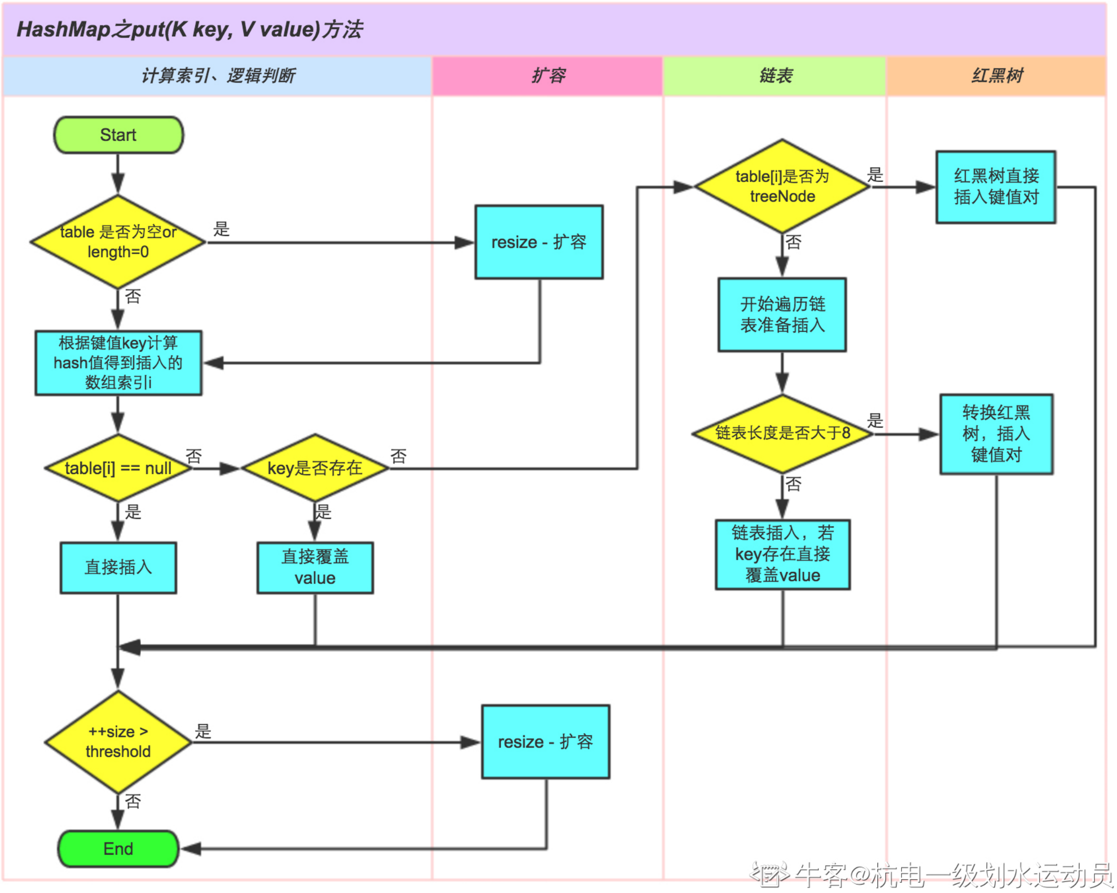
3）HashMap 对Null key做了单独存储，且value值可为null；HashTable不支持Null key Null value；

4）HashMap默认初始容量16，HashTable为11，扩容HashMap为两倍，HashTable为2n+1；

5）HashMap有红黑树，HashTable无该机制；

5）1.8 hashmap底层put过程 get过程

put



get:

1）判断哈希表是否为空、根据(hash & n-1) 判断对应桶是否为空

2）检查第一个元素是否相等

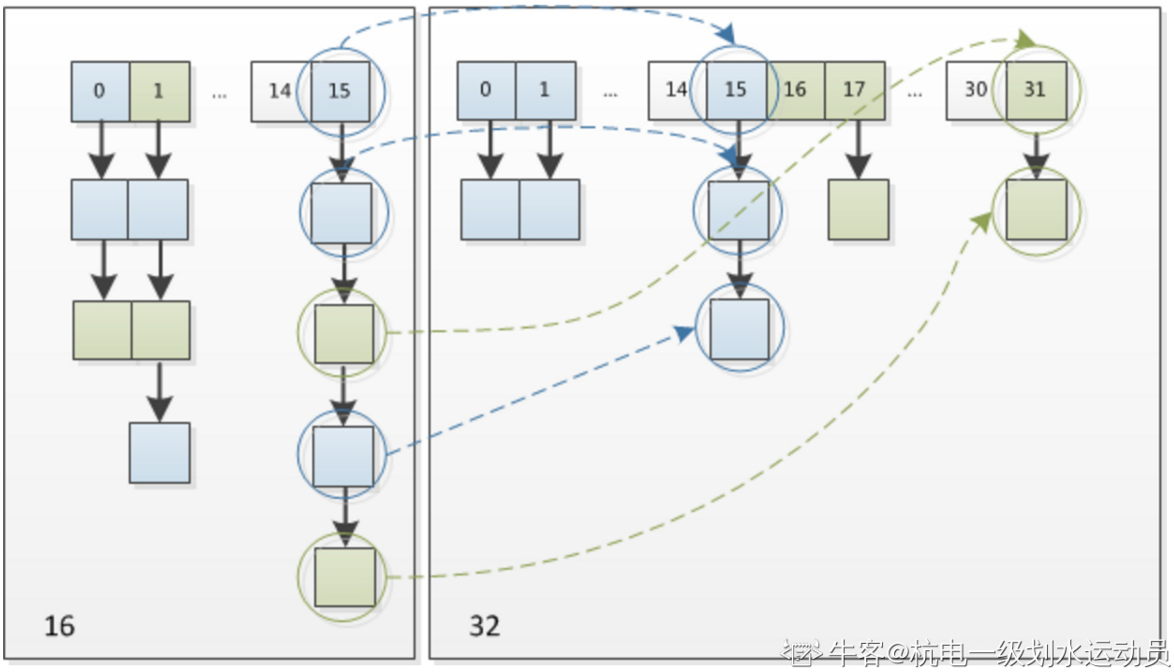
3）判断当前是否是树形还是链表，树形则进入树形搜索

4）链表则进行遍历检查

6）说一下hashmap扩容机制

1）当键值对个数size大于阈值thr时，将会扩容，将容量和阈值变为原来的两倍

2）然后，遍历原哈希表的所有元素，并且重新哈希到新表相应的桶中



7）Linkedhashmap，treemap为什么有序，查找时间复杂度

8）hashmap和b树有何异同，为什么数据库一般使用b树来当索引

待看完Mysql来填坑

9）1.8和1.7有什么不一样

1）hash计算1.7四次扰动，1.8直接高16位传递到低16位

2）1.7采用头插法、1.8保持相对顺序采用尾插

3）1.7采用数组+链表，1.8加入红黑树设计用来托底

不同点	JDK 1.7	JDK 1.8
存储结构	数组 + 链表	数组 + 链表 + 红黑树
初始化方式	单独函数：`inflateTable()`	直接集成到了扩容函数 `resize()` 中
hash值计算方式	扰动处理 = 9次扰动 = 4次位运算 + 5次异或运算	扰动处理 = 2次扰动 = 1次位运算 + 1次异或运算
存放数据的规则	无冲突时，存放数组；冲突时，存放链表	无冲突时，存放数组；冲突 & 链表长度 < 8：存放单链表；冲突 & 链表长度 > 8：树化并存放红黑树
插入数据方式	头插法（先讲原位置的数据移到后1位，再插入数据到该位置）	尾插法（直接插入到链表尾部/红黑树）
扩容后存储位置的计算方式	全部按照原来方法进行计算（即hashCode ->> 扰动函数 ->> (h&length-1)）	按照扩容后的规律计算（即扩容后的位置= 原位置 or 原位置 + 旧容量）

10）hashmap什么时候扩容，链表什么时候转为红黑树

键值对数量size大于thr，桶中元素大于8时转为红黑树，小于6时转回链表

11）HashMap concurrentHashMap

待看完多线程来填坑

12）concurrenthashmap，cas和synchronized

待看完多线程来填坑

```
package java.util;

import java.io.IOException;
```

```java
import java.io.InvalidObjectException;
import java.io.Serializable;
import java.lang.reflect.ParameterizedType;
import java.lang.reflect.Type;
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;

   //默认容量为16，容量大小需为2的次幂
    static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

   //最大容量为2的30次方
    static final int MAXIMUM_CAPACITY = 1 << 30;

   //默认装载因子是0.75
    static final float DEFAULT_LOAD_FACTOR = 0.75f;//为了空间和效率的平衡

   //当每个桶的容量超过8时，转为红黑树
    static final int TREEIFY_THRESHOLD = 8;

    static final int UNTREEIFY_THRESHOLD = 6;//当每个桶的容量小于6时，转为链表

    static final int MIN_TREEIFY_CAPACITY = 64;

    //内部节点类
    static class Node<K,V> implements Map.Entry<K,V> {
        final int hash;//散列值，用来确定桶下标
        final K key;
        V value;
        Node<K,V> next;

        Node(int hash, K key, V value, Node<K,V> next) {
            this.hash = hash;
            this.key = key;
            this.value = value;
            this.next = next;
        }

        public final K getKey()        { return key; }
        public final V getValue()      { return value; }
        public final String toString() { return key + "=" + value; }

        public final int hashCode() {
            return Objects.hashCode(key) ^ Objects.hashCode(value);
        }

        public final V setValue(V newValue) {
            V oldValue = value;
            value = newValue;
            return oldValue;
        }
```

```java
        public final boolean equals(Object o) {
            if (o == this)
                return true;
            if (o instanceof Map.Entry) {
                Map.Entry<?,?> e = (Map.Entry<?,?>)o;
                if (Objects.equals(key, e.getKey()) &&
                    Objects.equals(value, e.getValue()))
                    return true;
            }
            return false;
        }
    }

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


    static final int hash(Object key) {
        int h;
        return (key == null) ? 0 : (h = key.hashCode()) ^ (h >>> 16);
        //用关键词散列值的高16位和低16进行异或，来将高位信息传到低位，
        //以充分利用高位信息来进行桶的定位，防止在桶数量较小时，高位未利用；
    }



   //返回不小于指定容量的最小2次幂
    static final int tableSizeFor(int cap) {
        int n = cap - 1; //用或运算迭代将该数的所有二进制位变为1
        n |= n >>> 1;
        n |= n >>> 2;
        n |= n >>> 4;
        n |= n >>> 8;
        n |= n >>> 16;
        return (n < 0) ? 1 : (n >= MAXIMUM_CAPACITY) ? MAXIMUM_CAPACITY : n + 1;
    }

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

    //第一次使用时初始化，始终保持容量为二次幂，必要时扩容，可以为0来标记当前不需要的情况
    transient Node<K,V>[] table;

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

    //键值对数
    transient int size;


   //结构改变操作次数(增加，删除操作，修改value不算)，用在快速失败机制(多线程，A迭代先拷贝当前
modC,B修改      哈希表，则modCount改变，此时与A拷贝不一致)
    transient int modCount;


    int threshold;//扩容阈值 capacity * load factor

    final float loadFactor;//负载因子，可大于1

   //初始化，带容量和负载因子参数，此处可知负载因子没有小于1约束
```

```java
public HashMap(int initialCapacity, float loadFactor) {
    if (initialCapacity < 0)
        throw new IllegalArgumentException("Illegal initial capacity: " +
                                           initialCapacity);
    if (initialCapacity > MAXIMUM_CAPACITY)
        initialCapacity = MAXIMUM_CAPACITY;
    if (loadFactor <= 0 || Float.isNaN(loadFactor))
        throw new IllegalArgumentException("Illegal load factor: " +
                                           loadFactor);
    this.loadFactor = loadFactor;
    this.threshold = tableSizeFor(initialCapacity);
}


public HashMap(int initialCapacity) {
    this(initialCapacity, DEFAULT_LOAD_FACTOR);
}

//构造空的哈希表
public HashMap() {
    this.loadFactor = DEFAULT_LOAD_FACTOR; // all other fields defaulted
}


public HashMap(Map<? extends K, ? extends V> m) {
    this.loadFactor = DEFAULT_LOAD_FACTOR;
    putMapEntries(m, false);
}


final void putMapEntries(Map<? extends K, ? extends V> m, boolean evict) {
    int s = m.size();
    if (s > 0) {
        if (table == null) { // pre-size
            float ft = ((float)s / loadFactor) + 1.0F;
            int t = ((ft < (float)MAXIMUM_CAPACITY) ?
                     (int)ft : MAXIMUM_CAPACITY);
            if (t > threshold)
                threshold = tableSizeFor(t);
        }
        else if (s > threshold)
            resize();
        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);
        }
    }
}


public int size() {
    return size;
}


public boolean isEmpty() {
    return size == 0;
```

```java
    }


    //get方法过程
    public V get(Object key) {
        Node<K,V> e;
        return (e = getNode(hash(key), key)) == null ? null : e.value;
    }


    final Node<K,V> getNode(int hash, Object key) {
        Node<K,V>[] tab; Node<K,V> first, e; int n; K k;
        //判断哈希表是否不为空 且 对应桶是否为空
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (first = tab[(n - 1) & hash]) != null) {
            //不为空检查第一个元素是否等价
            if (first.hash == hash && // always check first node
                ((k = first.key) == key || (key != null && key.equals(k))))
                return first;
            //不等价，检查下一个元素
            if ((e = first.next) != null) {
                //判断是否是红黑树形式，是则进入树形模式
                if (first instanceof TreeNode)
                    return ((TreeNode<K,V>)first).getTreeNode(hash, key);
                //否则遍历链表进行检索
                do {
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        return e;
                } while ((e = e.next) != null);
            }
        }
        return null;
    }


    public boolean containsKey(Object key) {
        return getNode(hash(key), key) != null;
    }

    //keypoint put函数过程
    public V put(K key, V value) {
        return putVal(hash(key), key, value, false, true);
    }


    final V putVal(int hash, K key, V value, boolean onlyIfAbsent,
                   boolean evict) {
        Node<K,V>[] tab; Node<K,V> p; int n, i;
        //首先判断哈希表是否为空，或者容量为0，是则先扩容
        if ((tab = table) == null || (n = tab.length) == 0)
            n = (tab = resize()).length;
        //根据散列值计算桶下标，判断桶中是否有元素，没有元素则直接插入
        if ((p = tab[i = (n - 1) & hash]) == null)
            tab[i] = newNode(hash, key, value, null);

        else {
            Node<K,V> e; K k;
```

```java
            //判断桶首元素key是否等价，相等则值覆盖
            if (p.hash == hash &&
                ((k = p.key) == key || (key != null && key.equals(k))))
                e = p;
            //否则，判断该桶是链表还是红黑树形式，红黑树形式则进入树插入
            else if (p instanceof TreeNode)
                e = ((TreeNode<K,V>)p).putTreeVal(this, tab, hash, key, value);

            //否则，遍历链表找到key相等的元素进行值覆盖，或者插入链尾(尾插法 1.7是头插法)
            //插入后判断链表长度是否大于8(桶首下个位置开始计数，下标0-7此时为9个元素)，
            //是则转为红黑树
            else {
                for (int binCount = 0; ; ++binCount) {
                    if ((e = p.next) == null) {
                        p.next = newNode(hash, key, value, null);
                        if (binCount >= TREEIFY_THRESHOLD - 1) // -1 for 1st
                            treeifyBin(tab, hash);
                        break;
                    }
                    if (e.hash == hash &&
                        ((k = e.key) == key || (key != null && key.equals(k))))
                        break;
                    p = e;
                }
            }
            //直接返回原值
            if (e != null) { // existing mapping for key
                V oldValue = e.value;
                if (!onlyIfAbsent || oldValue == null)
                    e.value = value;
                afterNodeAccess(e);
                return oldValue;
            }
        }
        ++modCount;//此处可知，只有新增加、删除操作等结构操作才会增加修改值，覆盖原值已经直接
返回
        if (++size > threshold)
            resize();
        afterNodeInsertion(evict);
        return null;
    }

    //扩容过程
    final Node<K,V>[] resize() {
        Node<K,V>[] oldTab = table;
        int oldCap = (oldTab == null) ? 0 : oldTab.length;
        int oldThr = threshold;
        int newCap, newThr = 0;
        //若现有哈希表不为空或者容量为0
        if (oldCap > 0) {
            //现有容量大于等于最大容量上限，则仅将扩容阈值置为最大整数
            if (oldCap >= MAXIMUM_CAPACITY) {
                threshold = Integer.MAX_VALUE;
                return oldTab;
            }
            //否则，将容量扩大到现有容量的两倍，扩容阈值也翻倍
            else if ((newCap = oldCap << 1) < MAXIMUM_CAPACITY &&
                     oldCap >= DEFAULT_INITIAL_CAPACITY)
```

```
                newThr = oldThr << 1; // double threshold
        }
        //
        else if (oldThr > 0) // initial capacity was placed in threshold
            newCap = oldThr;
        else {               // zero initial threshold signifies using defaults
            newCap = DEFAULT_INITIAL_CAPACITY;
            newThr = (int)(DEFAULT_LOAD_FACTOR * DEFAULT_INITIAL_CAPACITY);
        }
        if (newThr == 0) {
            float ft = (float)newCap * loadFactor;
            newThr = (newCap < MAXIMUM_CAPACITY && ft < (float)MAXIMUM_CAPACITY
?
                      (int)ft : Integer.MAX_VALUE);
        }
        threshold = newThr;
        @SuppressWarnings({"rawtypes","unchecked"})
            Node<K,V>[] newTab = (Node<K,V>[])new Node[newCap];
        table = newTab;
        //拷贝和重哈希过程
        if (oldTab != null) {
            for (int j = 0; j < oldCap; ++j) {
                Node<K,V> e;
                //判断原数组当前位置是否为空
                if ((e = oldTab[j]) != null) {
                    oldTab[j] = null;
                    //判断当前桶是否不止一个元素
                    if (e.next == null)
                        newTab[e.hash & (newCap - 1)] = e;
                    //如果当前桶是红黑树形式，则进入红黑树过程
                    else if (e instanceof TreeNode)
                        ((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

                    //否则，将该桶中链表元素进行冲哈希
                    else { // preserve order
                        //建立两个指针，指向两个高、低桶位置链表的头
                        Node<K,V> loHead = null, loTail = null;
                        Node<K,V> hiHead = null, hiTail = null;
                        Node<K,V> next;
                        do {
                            next = e.next;

                            /*  由于1.8中计算桶的位置是用hash%n,由于容量是2^x = n,
                            *因此hash%n运算 = (n - 1) & hash运算，而后面这个运算
                            *相当于取了哈希值的二进制的x位，扩容后new_n = n*2 =
2^(x+1),
                            *因此，相当于取了哈希值的x+1位，故只需要判断当前哈希值的高位是
否
                            *为1，即可判断，该元素是落在高位桶，还是低位桶，而且可知两个桶
的
                            *下标之差为n;
                            *Exp: n = 4 = (100) => hash &   (011),
                            * new_n = 8 = (1000) => hash & (0111);
                            */
                            //如果当前元素的哈希值与上旧容量为0，插入低位链表
                            if ((e.hash & oldCap) == 0) {
                                if (loTail == null)
                                    loHead = e;
```

```java
                                else
                                    loTail.next = e;
                                loTail = e;
                            }
                            //否则插入高位链表
                            else {
                                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;
                            newTab[j + oldCap] = hiHead;
                        }
                    }
                }
            }
        }
        return newTab;
    }

    /**
     * Replaces all linked nodes in bin at index for given hash unless
     * table is too small, in which case resizes instead.
     */
    final void treeifyBin(Node<K,V>[] tab, int hash) {
        int n, index; Node<K,V> e;
        if (tab == null || (n = tab.length) < MIN_TREEIFY_CAPACITY)
            resize();
        else if ((e = tab[index = (n - 1) & hash]) != null) {
            TreeNode<K,V> hd = null, tl = null;
            do {
                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) {
        Node<K,V>[] tab; Node<K,V> p; int n, index;
        if ((tab = table) != null && (n = tab.length) > 0 &&
            (p = tab[index = (n - 1) & hash]) != null) {
            Node<K,V> node = null, e; K k; V v;
            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)
                    node = ((TreeNode<K,V>)p).getTreeNode(hash, key);
                else {
                    do {
                        if (e.hash == hash &&
                            ((k = e.key) == key ||
                             (key != null && key.equals(k)))) {
                            node = e;
                            break;
                        }
                        p = e;
                    } while ((e = e.next) != null);
                }
            }
            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);
                else if (node == p)
                    tab[index] = node.next;
                else
                    p.next = node.next;
                ++modCount;
                --size;
                afterNodeRemoval(node);
                return node;
            }
        }
        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,
```

```java
     * 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(); }
```

```java
        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;
    }

    @Override
    public V computeIfAbsent(K key,
                             Function<? super K, ? extends V> mappingFunction) {
        if (mappingFunction == 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;
            if (old != null && (oldValue = old.value) != null) {
                afterNodeAccess(old);
                return oldValue;
            }
        }
        V v = mappingFunction.apply(key);
```

```java
        if (v == null) {
            return null;
        } else if (old != null) {
            old.value = v;
            afterNodeAccess(old);
            return v;
        }
        else if (t != null)
            t.putTreeVal(this, tab, hash, key, v);
        else {
            tab[i] = newNode(hash, key, v, first);
            if (binCount >= TREEIFY_THRESHOLD - 1)
                treeifyBin(tab, hash);
        }
        ++modCount;
        ++size;
        afterNodeInsertion(true);
        return v;
    }

    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;
    }

    @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 >= 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 {
```

```java
                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 >= 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 :
            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);
    }

    /**
     * Reconstitute the {@code HashMap} instance from a stream (i.e.,
     * deserialize it).
     */
    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 < DEFAULT_INITIAL_CAPACITY) ?
                       DEFAULT_INITIAL_CAPACITY :
                       (fc >= MAXIMUM_CAPACITY) ?
                       MAXIMUM_CAPACITY :
                       tableSizeFor((int)fc));
            float ft = (float)cap * lf;
            threshold = ((cap < MAXIMUM_CAPACITY && ft < 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);
```

```java
            }
        }
    }

    /* ------------------------------------------------------------ */
    // 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; }
    }
```

```java
    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

    static class HashMapSpliterator<K,V> {
        final HashMap<K,V> map;
        Node<K,V> current;          // current node
        int index;                  // current index, modified on advance/split
        int fence;                  // one past last index
        int est;                    // size estimate
        int expectedModCount;       // for comodification checks

        HashMapSpliterator(HashMap<K,V> m, int origin,
                           int fence, int est,
                           int expectedModCount) {
            this.map = m;
            this.index = origin;
            this.fence = fence;
            this.est = est;
            this.expectedModCount = expectedModCount;
        }

        final int getFence() { // initialize fence and size on first use
            int hi;
            if ((hi = fence) < 0) {
                HashMap<K,V> m = map;
                est = m.size;
                expectedModCount = m.modCount;
                Node<K,V>[] tab = m.table;
                hi = fence = (tab == null) ? 0 : tab.length;
            }
            return hi;
        }

        public final long estimateSize() {
            getFence(); // force init
            return (long) est;
        }
    }

    static final class KeySpliterator<K,V>
        extends HashMapSpliterator<K,V>
        implements Spliterator<K> {
        KeySpliterator(HashMap<K,V> m, int origin, int fence, int est,
                       int expectedModCount) {
            super(m, origin, fence, est, expectedModCount);
        }

        public KeySpliterator<K,V> trySplit() {
```

```java
            int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
            return (lo >= mid || current != null) ? null :
                new KeySpliterator<>(map, lo, index = mid, est >>>= 1,
                                        expectedModCount);
        }

        public void forEachRemaining(Consumer<? super K> action) {
            int i, hi, mc;
            if (action == null)
                throw new NullPointerException();
            HashMap<K,V> m = map;
            Node<K,V>[] tab = m.table;
            if ((hi = fence) < 0) {
                mc = expectedModCount = m.modCount;
                hi = fence = (tab == null) ? 0 : tab.length;
            }
            else
                mc = expectedModCount;
            if (tab != null && tab.length >= hi &&
                (i = index) >= 0 && (i < (index = hi) || current != null)) {
                Node<K,V> p = current;
                current = null;
                do {
                    if (p == null)
                        p = tab[i++];
                    else {
                        action.accept(p.key);
                        p = p.next;
                    }
                } while (p != null || i < hi);
                if (m.modCount != mc)
                    throw new ConcurrentModificationException();
            }
        }

        public boolean tryAdvance(Consumer<? super K> action) {
            int hi;
            if (action == null)
                throw new NullPointerException();
            Node<K,V>[] tab = map.table;
            if (tab != null && tab.length >= (hi = getFence()) && index >= 0) {
                while (current != null || index < hi) {
                    if (current == null)
                        current = tab[index++];
                    else {
                        K k = current.key;
                        current = current.next;
                        action.accept(k);
                        if (map.modCount != expectedModCount)
                            throw new ConcurrentModificationException();
                        return true;
                    }
                }
            }
            return false;
        }

        public int characteristics() {
```

```
            return (fence < 0 || est == map.size ? Spliterator.SIZED : 0) |
                Spliterator.DISTINCT;
        }
    }

    static final class ValueSpliterator<K,V>
        extends HashMapSpliterator<K,V>
        implements Spliterator<V> {
        ValueSpliterator(HashMap<K,V> m, int origin, int fence, int est,
                         int expectedModCount) {
            super(m, origin, fence, est, expectedModCount);
        }

        public ValueSpliterator<K,V> trySplit() {
            int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
            return (lo >= mid || current != null) ? null :
                new ValueSpliterator<>(map, lo, index = mid, est >>>= 1,
                                          expectedModCount);
        }

        public void forEachRemaining(Consumer<? super V> action) {
            int i, hi, mc;
            if (action == null)
                throw new NullPointerException();
            HashMap<K,V> m = map;
            Node<K,V>[] tab = m.table;
            if ((hi = fence) < 0) {
                mc = expectedModCount = m.modCount;
                hi = fence = (tab == null) ? 0 : tab.length;
            }
            else
                mc = expectedModCount;
            if (tab != null && tab.length >= hi &&
                (i = index) >= 0 && (i < (index = hi) || current != null)) {
                Node<K,V> p = current;
                current = null;
                do {
                    if (p == null)
                        p = tab[i++];
                    else {
                        action.accept(p.value);
                        p = p.next;
                    }
                } while (p != null || i < hi);
                if (m.modCount != mc)
                    throw new ConcurrentModificationException();
            }
        }

        public boolean tryAdvance(Consumer<? super V> action) {
            int hi;
            if (action == null)
                throw new NullPointerException();
            Node<K,V>[] tab = map.table;
            if (tab != null && tab.length >= (hi = getFence()) && index >= 0) {
                while (current != null || index < hi) {
                    if (current == null)
                        current = tab[index++];
```

```java
                    else {
                        V v = current.value;
                        current = current.next;
                        action.accept(v);
                        if (map.modCount != expectedModCount)
                            throw new ConcurrentModificationException();
                        return true;
                    }
                }
            }
            return false;
        }

        public int characteristics() {
            return (fence < 0 || est == map.size ? Spliterator.SIZED : 0);
        }
    }

    static final class EntrySpliterator<K,V>
        extends HashMapSpliterator<K,V>
        implements Spliterator<Map.Entry<K,V>> {
        EntrySpliterator(HashMap<K,V> m, int origin, int fence, int est,
                         int expectedModCount) {
            super(m, origin, fence, est, expectedModCount);
        }

        public EntrySpliterator<K,V> trySplit() {
            int hi = getFence(), lo = index, mid = (lo + hi) >>> 1;
            return (lo >= mid || current != null) ? null :
                new EntrySpliterator<>(map, lo, index = mid, est >>>= 1,
                                          expectedModCount);
        }

        public void forEachRemaining(Consumer<? super Map.Entry<K,V>> action) {
            int i, hi, mc;
            if (action == null)
                throw new NullPointerException();
            HashMap<K,V> m = map;
            Node<K,V>[] tab = m.table;
            if ((hi = fence) < 0) {
                mc = expectedModCount = m.modCount;
                hi = fence = (tab == null) ? 0 : tab.length;
            }
            else
                mc = expectedModCount;
            if (tab != null && tab.length >= hi &&
                (i = index) >= 0 && (i < (index = hi) || current != null)) {
                Node<K,V> p = current;
                current = null;
                do {
                    if (p == null)
                        p = tab[i++];
                    else {
                        action.accept(p);
                        p = p.next;
                    }
                } while (p != null || i < hi);
                if (m.modCount != mc)
```

```java
                    throw new ConcurrentModificationException();
            }
        }

        public boolean tryAdvance(Consumer<? super Map.Entry<K,V>> action) {
            int hi;
            if (action == null)
                throw new NullPointerException();
            Node<K,V>[] tab = map.table;
            if (tab != null && tab.length >= (hi = getFence()) && index >= 0) {
                while (current != null || index < hi) {
                    if (current == null)
                        current = tab[index++];
                    else {
                        Node<K,V> e = current;
                        current = current.next;
                        action.accept(e);
                        if (map.modCount != expectedModCount)
                            throw new ConcurrentModificationException();
                        return true;
                    }
                }
            }
            return false;
        }

        public int characteristics() {
            return (fence < 0 || est == map.size ? Spliterator.SIZED : 0) |
                Spliterator.DISTINCT;
        }
    }

    /* ------------------------------------------------------------ */
    // 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);
    }
```

```java
    // 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) {
            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);
                }
            }
        }
    }

    /* ------------------------------------------------------------ */
    // Tree bins

    /**
     * Entry for Tree bins. Extends LinkedHashMap.Entry (which in turn
     * extends Node) so can be used as extension of either regular or
     * linked node.
     */
    static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
        TreeNode<K,V> parent;  // red-black tree links
        TreeNode<K,V> left;
        TreeNode<K,V> right;
        TreeNode<K,V> prev;    // needed to unlink next upon deletion
        boolean red;
        TreeNode(int hash, K key, V val, Node<K,V> next) {
            super(hash, key, val, next);
        }

        /**
         * Returns root of tree containing this node.
         */
```

```java
        final TreeNode<K,V> root() {
            for (TreeNode<K,V> r = this, p;;) {
                if ((p = r.parent) == null)
                    return r;
                r = p;
            }
        }

        /**
         * Ensures that the given root is the first node of its bin.
         */
        static <K,V> void moveRootToFront(Node<K,V>[] tab, TreeNode<K,V> root) {
            int n;
            if (root != null && tab != null && (n = tab.length) > 0) {
                int index = (n - 1) & root.hash;
                TreeNode<K,V> first = (TreeNode<K,V>)tab[index];
                if (root != first) {
                    Node<K,V> rn;
                    tab[index] = root;
                    TreeNode<K,V> rp = root.prev;
                    if ((rn = root.next) != null)
                        ((TreeNode<K,V>)rn).prev = rp;
                    if (rp != null)
                        rp.next = rn;
                    if (first != null)
                        first.prev = root;
                    root.next = first;
                    root.prev = null;
                }
                assert checkInvariants(root);
            }
        }

        /**
         * Finds the node starting at root p with the given hash and key.
         * The kc argument caches comparableClassFor(key) upon first use
         * comparing keys.
         */
        final TreeNode<K,V> find(int h, Object k, Class<?> kc) {
            TreeNode<K,V> p = this;
            do {
                int ph, dir; K pk;
                TreeNode<K,V> pl = p.left, pr = p.right, q;
                if ((ph = p.hash) > h)
                    p = pl;
                else if (ph < h)
                    p = pr;
                else if ((pk = p.key) == k || (k != null && k.equals(pk)))
                    return p;
                else if (pl == null)
                    p = pr;
                else if (pr == null)
                    p = pl;
                else if ((kc != null ||
                          (kc = comparableClassFor(k)) != null) &&
                         (dir = compareComparables(kc, k, pk)) != 0)
                    p = (dir < 0) ? pl : pr;
                else if ((q = pr.find(h, k, kc)) != null)
```

```java
                return q;
            else
                p = pl;
        } while (p != null);
        return null;
    }

    /**
     * Calls find for root node.
     */
    final TreeNode<K,V> getTreeNode(int h, Object k) {
        return ((parent != null) ? root() : this).find(h, k, null);
    }

    /**
     * Tie-breaking utility for ordering insertions when equal
     * hashCodes and non-comparable. We don't require a total
     * order, just a consistent insertion rule to maintain
     * equivalence across rebalancings. Tie-breaking further than
     * necessary simplifies testing a bit.
     */
    static int tieBreakOrder(Object a, Object b) {
        int d;
        if (a == null || b == null ||
            (d = a.getClass().getName().
             compareTo(b.getClass().getName())) == 0)
            d = (System.identityHashCode(a) <= System.identityHashCode(b) ?
                 -1 : 1);
        return d;
    }

    /**
     * Forms tree of the nodes linked from this node.
     * @return root of tree
     */
    final void treeify(Node<K,V>[] tab) {
        TreeNode<K,V> root = null;
        for (TreeNode<K,V> x = this, next; x != null; x = next) {
            next = (TreeNode<K,V>)x.next;
            x.left = x.right = null;
            if (root == null) {
                x.parent = null;
                x.red = false;
                root = x;
            }
            else {
                K k = x.key;
                int h = x.hash;
                Class<?> kc = null;
                for (TreeNode<K,V> p = root;;) {
                    int dir, ph;
                    K pk = p.key;
                    if ((ph = p.hash) > h)
                        dir = -1;
                    else if (ph < h)
                        dir = 1;
                    else if ((kc == null &&
                              (kc = comparableClassFor(k)) == null) ||
```

```java
                             (dir = compareComparables(kc, k, pk)) == 0)
                        dir = tieBreakOrder(k, pk);

                    TreeNode<K,V> xp = p;
                    if ((p = (dir <= 0) ? p.left : p.right) == null) {
                        x.parent = xp;
                        if (dir <= 0)
                            xp.left = x;
                        else
                            xp.right = x;
                        root = balanceInsertion(root, x);
                        break;
                    }
                }
            }
        }
        moveRootToFront(tab, root);
    }

    /**
     * Returns a list of non-TreeNodes replacing those linked from
     * this node.
     */
    final Node<K,V> untreeify(HashMap<K,V> map) {
        Node<K,V> hd = null, tl = null;
        for (Node<K,V> q = this; q != null; q = q.next) {
            Node<K,V> p = map.replacementNode(q, null);
            if (tl == null)
                hd = p;
            else
                tl.next = p;
            tl = p;
        }
        return hd;
    }

    /**
     * Tree version of putVal.
     */
    final TreeNode<K,V> putTreeVal(HashMap<K,V> map, Node<K,V>[] tab,
                                   int h, K k, V v) {
        Class<?> kc = null;
        boolean searched = false;
        TreeNode<K,V> root = (parent != null) ? root() : this;
        for (TreeNode<K,V> p = root;;) {
            int dir, ph; K pk;
            if ((ph = p.hash) > h)
                dir = -1;
            else if (ph < h)
                dir = 1;
            else if ((pk = p.key) == k || (k != null && k.equals(pk)))
                return p;
            else if ((kc == null &&
                      (kc = comparableClassFor(k)) == null) ||
                     (dir = compareComparables(kc, k, pk)) == 0) {
                if (!searched) {
                    TreeNode<K,V> q, ch;
                    searched = true;
```

```java
                if (((ch = p.left) != null &&
                     (q = ch.find(h, k, kc)) != null) ||
                    ((ch = p.right) != null &&
                     (q = ch.find(h, k, kc)) != null))
                    return q;
            }
            dir = tieBreakOrder(k, pk);
        }

        TreeNode<K,V> xp = p;
        if ((p = (dir <= 0) ? p.left : p.right) == null) {
            Node<K,V> xpn = xp.next;
            TreeNode<K,V> x = map.newTreeNode(h, k, v, xpn);
            if (dir <= 0)
                xp.left = x;
            else
                xp.right = x;
            xp.next = x;
            x.parent = x.prev = xp;
            if (xpn != null)
                ((TreeNode<K,V>)xpn).prev = x;
            moveRootToFront(tab, balanceInsertion(root, x));
            return null;
        }
    }
}

/**
 * Removes the given node, that must be present before this call.
 * This is messier than typical red-black deletion code because we
 * cannot swap the contents of an interior node with a leaf
 * successor that is pinned by "next" pointers that are accessible
 * independently during traversal. So instead we swap the tree
 * linkages. If the current tree appears to have too few nodes,
 * the bin is converted back to a plain bin. (The test triggers
 * somewhere between 2 and 6 nodes, depending on tree structure).
 */
final void removeTreeNode(HashMap<K,V> map, Node<K,V>[] tab,
                          boolean movable) {
    int n;
    if (tab == null || (n = tab.length) == 0)
        return;
    int index = (n - 1) & hash;
    TreeNode<K,V> first = (TreeNode<K,V>)tab[index], root = first, rl;
    TreeNode<K,V> succ = (TreeNode<K,V>)next, pred = prev;
    if (pred == null)
        tab[index] = first = succ;
    else
        pred.next = succ;
    if (succ != null)
        succ.prev = pred;
    if (first == null)
        return;
    if (root.parent != null)
        root = root.root();
    if (root == null || root.right == null ||
        (rl = root.left) == null || rl.left == null) {
        tab[index] = first.untreeify(map);  // too small
```

```java
        return;
    }
    TreeNode<K,V> p = this, pl = left, pr = right, replacement;
    if (pl != null && pr != null) {
        TreeNode<K,V> s = pr, sl;
        while ((sl = s.left) != null) // find successor
            s = sl;
        boolean c = s.red; s.red = p.red; p.red = c; // swap colors
        TreeNode<K,V> sr = s.right;
        TreeNode<K,V> pp = p.parent;
        if (s == pr) { // p was s's direct parent
            p.parent = s;
            s.right = p;
        }
        else {
            TreeNode<K,V> sp = s.parent;
            if ((p.parent = sp) != null) {
                if (s == sp.left)
                    sp.left = p;
                else
                    sp.right = p;
            }
            if ((s.right = pr) != null)
                pr.parent = s;
        }
        p.left = null;
        if ((p.right = sr) != null)
            sr.parent = p;
        if ((s.left = pl) != null)
            pl.parent = s;
        if ((s.parent = pp) == null)
            root = s;
        else if (p == pp.left)
            pp.left = s;
        else
            pp.right = s;
        if (sr != null)
            replacement = sr;
        else
            replacement = p;
    }
    else if (pl != null)
        replacement = pl;
    else if (pr != null)
        replacement = pr;
    else
        replacement = p;
    if (replacement != p) {
        TreeNode<K,V> pp = replacement.parent = p.parent;
        if (pp == null)
            root = replacement;
        else if (p == pp.left)
            pp.left = replacement;
        else
            pp.right = replacement;
        p.left = p.right = p.parent = null;
    }
```

```java
            TreeNode<K,V> r = p.red ? root : balanceDeletion(root, replacement);

            if (replacement == p) {  // detach
                TreeNode<K,V> pp = p.parent;
                p.parent = null;
                if (pp != null) {
                    if (p == pp.left)
                        pp.left = null;
                    else if (p == pp.right)
                        pp.right = null;
                }
            }
            if (movable)
                moveRootToFront(tab, r);
        }

        /**
         * Splits nodes in a tree bin into lower and upper tree bins,
         * or untreeifies if now too small. Called only from resize;
         * see above discussion about split bits and indices.
         *
         * @param map the map
         * @param tab the table for recording bin heads
         * @param index the index of the table being split
         * @param bit the bit of hash to split on
         */
        final void split(HashMap<K,V> map, Node<K,V>[] tab, int index, int bit)
{
            TreeNode<K,V> b = this;
            // Relink into lo and hi lists, preserving order
            TreeNode<K,V> loHead = null, loTail = null;
            TreeNode<K,V> hiHead = null, hiTail = null;
            int lc = 0, hc = 0;
            for (TreeNode<K,V> e = b, next; e != null; e = next) {
                next = (TreeNode<K,V>)e.next;
                e.next = null;
                if ((e.hash & bit) == 0) {
                    if ((e.prev = loTail) == null)
                        loHead = e;
                    else
                        loTail.next = e;
                    loTail = e;
                    ++lc;
                }
                else {
                    if ((e.prev = hiTail) == null)
                        hiHead = e;
                    else
                        hiTail.next = e;
                    hiTail = e;
                    ++hc;
                }
            }

            if (loHead != null) {
                if (lc <= UNTREEIFY_THRESHOLD)
                    tab[index] = loHead.untreeify(map);
                else {
```

```java
                tab[index] = loHead;
                if (hiHead != null) // (else is already treeified)
                    loHead.treeify(tab);
            }
        }
        if (hiHead != null) {
            if (hc <= UNTREEIFY_THRESHOLD)
                tab[index + bit] = hiHead.untreeify(map);
            else {
                tab[index + bit] = hiHead;
                if (loHead != null)
                    hiHead.treeify(tab);
            }
        }
    }

    /* ------------------------------------------------------------ */
    // Red-black tree methods, all adapted from CLR

    static <K,V> TreeNode<K,V> rotateLeft(TreeNode<K,V> root,
                                          TreeNode<K,V> p) {
        TreeNode<K,V> r, pp, rl;
        if (p != null && (r = p.right) != null) {
            if ((rl = p.right = r.left) != null)
                rl.parent = p;
            if ((pp = r.parent = p.parent) == null)
                (root = r).red = false;
            else if (pp.left == p)
                pp.left = r;
            else
                pp.right = r;
            r.left = p;
            p.parent = r;
        }
        return root;
    }

    static <K,V> TreeNode<K,V> rotateRight(TreeNode<K,V> root,
                                           TreeNode<K,V> p) {
        TreeNode<K,V> l, pp, lr;
        if (p != null && (l = p.left) != null) {
            if ((lr = p.left = l.right) != null)
                lr.parent = p;
            if ((pp = l.parent = p.parent) == null)
                (root = l).red = false;
            else if (pp.right == p)
                pp.right = l;
            else
                pp.left = l;
            l.right = p;
            p.parent = l;
        }
        return root;
    }

    static <K,V> TreeNode<K,V> balanceInsertion(TreeNode<K,V> root,
                                                TreeNode<K,V> x) {
        x.red = true;
```

```java
        for (TreeNode<K,V> xp, xpp, xppl, xppr;;) {
            if ((xp = x.parent) == null) {
                x.red = false;
                return x;
            }
            else if (!xp.red || (xpp = xp.parent) == null)
                return root;
            if (xp == (xppl = xpp.left)) {
                if ((xppr = xpp.right) != null && xppr.red) {
                    xppr.red = false;
                    xp.red = false;
                    xpp.red = true;
                    x = xpp;
                }
                else {
                    if (x == xp.right) {
                        root = rotateLeft(root, x = xp);
                        xpp = (xp = x.parent) == null ? null : xp.parent;
                    }
                    if (xp != null) {
                        xp.red = false;
                        if (xpp != null) {
                            xpp.red = true;
                            root = rotateRight(root, xpp);
                        }
                    }
                }
            }
            else {
                if (xppl != null && xppl.red) {
                    xppl.red = false;
                    xp.red = false;
                    xpp.red = true;
                    x = xpp;
                }
                else {
                    if (x == xp.left) {
                        root = rotateRight(root, x = xp);
                        xpp = (xp = x.parent) == null ? null : xp.parent;
                    }
                    if (xp != null) {
                        xp.red = false;
                        if (xpp != null) {
                            xpp.red = true;
                            root = rotateLeft(root, xpp);
                        }
                    }
                }
            }
        }
    }

    static <K,V> TreeNode<K,V> balanceDeletion(TreeNode<K,V> root,
                                               TreeNode<K,V> x) {
        for (TreeNode<K,V> xp, xpl, xpr;;)  {
            if (x == null || x == root)
                return root;
            else if ((xp = x.parent) == null) {
```

```java
                x.red = false;
                return x;
            }
            else if (x.red) {
                x.red = false;
                return root;
            }
            else if ((xpl = xp.left) == x) {
                if ((xpr = xp.right) != null && xpr.red) {
                    xpr.red = false;
                    xp.red = true;
                    root = rotateLeft(root, xp);
                    xpr = (xp = x.parent) == null ? null : xp.right;
                }
                if (xpr == null)
                    x = xp;
                else {
                    TreeNode<K,V> sl = xpr.left, sr = xpr.right;
                    if ((sr == null || !sr.red) &&
                        (sl == null || !sl.red)) {
                        xpr.red = true;
                        x = xp;
                    }
                    else {
                        if (sr == null || !sr.red) {
                            if (sl != null)
                                sl.red = false;
                            xpr.red = true;
                            root = rotateRight(root, xpr);
                            xpr = (xp = x.parent) == null ?
                                null : xp.right;
                        }
                        if (xpr != null) {
                            xpr.red = (xp == null) ? false : xp.red;
                            if ((sr = xpr.right) != null)
                                sr.red = false;
                        }
                        if (xp != null) {
                            xp.red = false;
                            root = rotateLeft(root, xp);
                        }
                        x = root;
                    }
                }
            }
            else { // symmetric
                if (xpl != null && xpl.red) {
                    xpl.red = false;
                    xp.red = true;
                    root = rotateRight(root, xp);
                    xpl = (xp = x.parent) == null ? null : xp.left;
                }
                if (xpl == null)
                    x = xp;
                else {
                    TreeNode<K,V> sl = xpl.left, sr = xpl.right;
                    if ((sl == null || !sl.red) &&
                        (sr == null || !sr.red)) {
```

```java
                            xpl.red = true;
                            x = xp;
                        }
                        else {
                            if (sl == null || !sl.red) {
                                if (sr != null)
                                    sr.red = false;
                                xpl.red = true;
                                root = rotateLeft(root, xpl);
                                xpl = (xp = x.parent) == null ?
                                    null : xp.left;
                            }
                            if (xpl != null) {
                                xpl.red = (xp == null) ? false : xp.red;
                                if ((sl = xpl.left) != null)
                                    sl.red = false;
                            }
                            if (xp != null) {
                                xp.red = false;
                                root = rotateRight(root, xp);
                            }
                            x = root;
                        }
                    }
                }
            }
        }

        /**
         * Recursive invariant check
         */
        static <K,V> boolean checkInvariants(TreeNode<K,V> t) {
            TreeNode<K,V> tp = t.parent, tl = t.left, tr = t.right,
                tb = t.prev, tn = (TreeNode<K,V>)t.next;
            if (tb != null && tb.next != t)
                return false;
            if (tn != null && tn.prev != t)
                return false;
            if (tp != null && t != tp.left && t != tp.right)
                return false;
            if (tl != null && (tl.parent != t || tl.hash > t.hash))
                return false;
            if (tr != null && (tr.parent != t || tr.hash < t.hash))
                return false;
            if (t.red && tl != null && tl.red && tr != null && tr.red)
                return false;
            if (tl != null && !checkInvariants(tl))
                return false;
            if (tr != null && !checkInvariants(tr))
                return false;
            return true;
        }
    }

}
```