

List和Set是存储单列数据的集合，Map是存储键值对这样的双列数据的集合；

List中存储的数据是有顺序的，并且值允许重复；

Set中存储的数据是无顺序的，并且不允许重复。

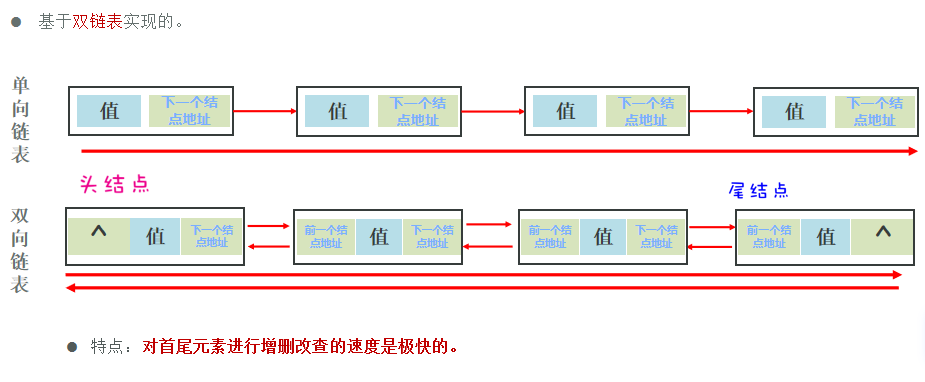
Map中存储的数据是无序的，它的键是不允许重复的，但是值是允许重复的；

**List接口有三个实现类（有序可重复）：**

查询多：ArrayList  
增删多：LinkedList  
如果你知道是List，但是不知道是哪个List，就用ArrayList。

1.1 LinkedList

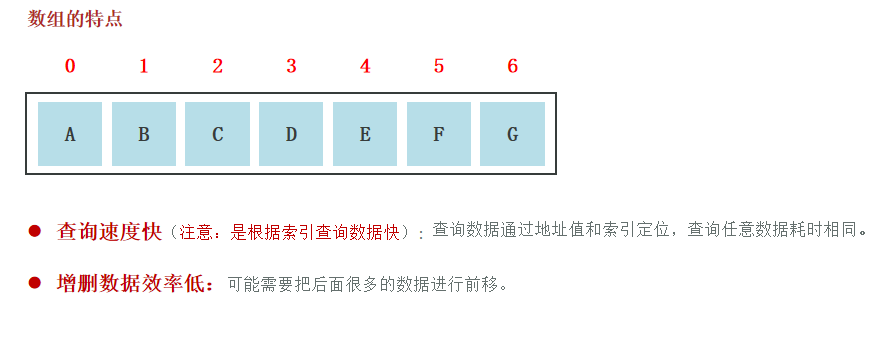
基于双链表实现适合在头部尾部增删元素，线程不安全，链表内存是散列的，增删快，查找慢；





1.2 ArrayList

基于数组实现，非线程安全，效率高，增删慢，查找快；





1.3 Vector

基于数组实现，线程安全，效率低，查询快，增删慢；

**Map接口有四个实现类：**

**Map方法**



2.1 HashMap

基于 hash 表的 Map 接口实现，非线程安全，高效，支持 null 值和 null

键；无序、不重复、无索引；

2.2 LinkedHashMap

是 HashMap 的一个子类，保存了记录的插入顺序；有序、不重复、无索引。

2.3 TreeMap 接口

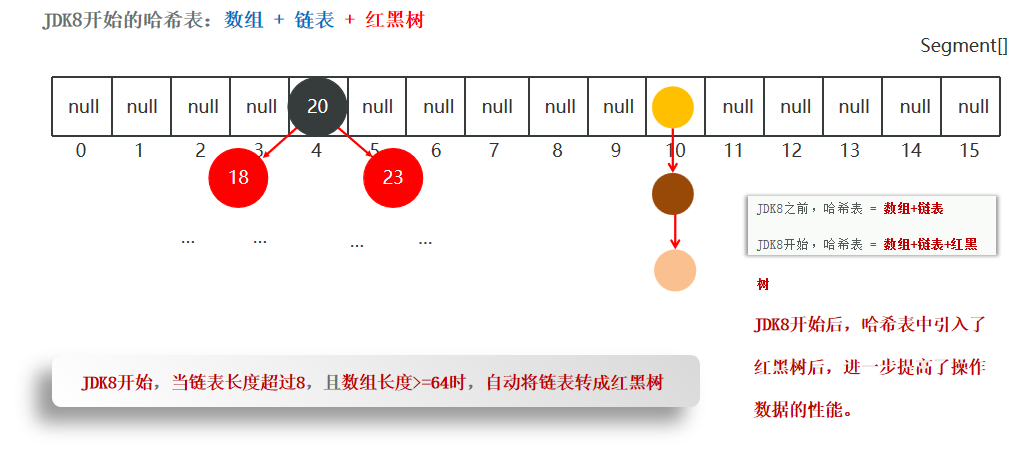
TreeMap，能够把它保存的记录根据键排序，默认是键值的升序排序,不重复、无索引。

**Set接口（无序，唯一）有两个实现类：**

是：TreeSet或LinkedHashSet  
否：HashSet  
如果你知道是Set，但是不知道是哪个Set，就用HashSet。

3.1 HashSet

底层是由 Hash Map哈希表 实现，不允许集合中有重复的值，使用该方式时需要重写 equals()和 hash Code()方法；



初始状态

容量：16

阈值：16 \* 0.75 = 12

当插入第 13 个元素时触发扩容。

第一次扩容

新容量：32

新阈值：32 \* 0.75 = 24

当插入第 25 个元素时再次扩容。

第二次扩容

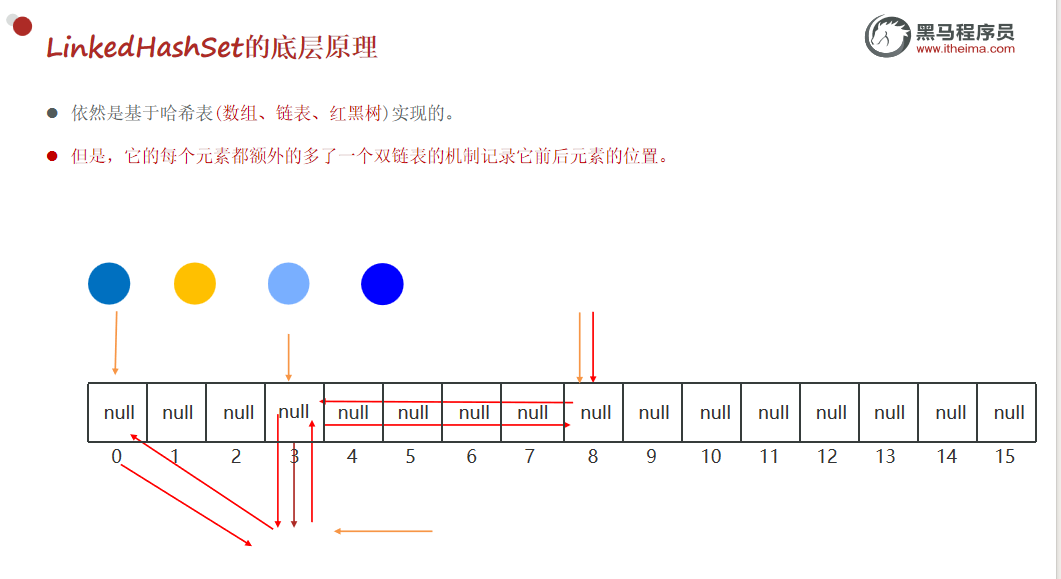
新容量：64

新阈值：64 \* 0.75 = 48

如此类推，每次扩容容量都会翻倍。

3.2 LinkedHashSet

继承于 HashSet，同时又基于 LinkedHashMap 来进行实现，底层使用的是 LinkedHashMap

用链表保证元素有序，用哈希表保证元素唯一

3.3Treeset

不重复、无索引、可排序（默认升序排序 ，按照元素的大小，由小到大排序）

底层是基于红黑树实现的排序。

ArreyList扩容实现逻辑

public ArrayList() {

this.elementData = DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA;

}初始为空数组，第一次添加元素时会扩容到 DEFAULT\_CAPACITY=10。

private Object[] grow(int minCapacity) {

int oldCapacity = elementData.length;

if (oldCapacity > 0 || elementData != DEFAULTCAPACITY\_EMPTY\_ELEMENTDATA) {

int newCapacity = ArraysSupport.newLength(oldCapacity,

minCapacity - oldCapacity, /\* minimum growth \*/

oldCapacity >> 1 /\* preferred growth \*/);

return elementData = Arrays.copyOf(elementData, newCapacity);

} else {

return elementData = new Object[Math.max(DEFAULT\_CAPACITY, minCapacity)];

}

}

public static int newLength(int oldLength, int minGrowth, int prefGrowth) {

// preconditions not checked because of inlining

// assert oldLength >= 0

// assert minGrowth > 0

推荐扩容量=（原容量+最小容量和推荐容量的最大值）推荐扩容量为 oldCapacity >> 1即原来数组的一半的长度

int prefLength = oldLength + Math.max(minGrowth, prefGrowth); // might overflow

if (0 < prefLength && prefLength <= SOFT\_MAX\_ARRAY\_LENGTH（这个值为0x7fffffff-8）) {

return prefLength;

//超过扩容最大值使用hugeLength

} else {

// put code cold in a separate method

return hugeLength(oldLength, minGrowth);

}

}

Hashset底层实现逻辑

transient HashMap<E,Object> map;

所有元素都存储在 map 的 key 中。

你提供的代码是 Java 中 `HashMap` 的核心方法之一 —— `resize()` 和 `treeifyBin()`。它们分别负责：

1. \*\*扩容机制\*\*：当键值对数量超过阈值时，扩大底层数组并重新分配元素。

2. \*\*链表转红黑树\*\*：当某个桶中的链表长度过长时，将其转换为红黑树以提高查询效率。

我们将逐行详细解释这段代码的实现逻辑，并结合你的记忆知识说明其设计目的与规范。

---

## 一、函数签名与局部变量定义

```java

final Node<K,V>[] resize() {

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

int oldCap = (oldTab == null) ? 0 : oldTab.length;

int oldThr = threshold;

int newCap, newThr = 0;

```

### 变量说明：

- `oldTab`：指向当前使用的底层数组 [table](file://E:\javacode\week1\src\map.java#L37-L37)。

- `oldCap`：当前数组容量（即 `oldTab.length`）。

- `oldThr`：旧的扩容阈值（capacity \* load factor）。

- `newCap`：新的容量。

- `newThr`：新的扩容阈值。

---

## 二、判断当前是否需要扩容

### 1. 如果当前容量大于 0（已初始化）

```java

if (oldCap > 0) {

if (oldCap >= MAXIMUM\_CAPACITY) {

threshold = Integer.MAX\_VALUE;

return oldTab;

}

else if ((newCap = oldCap << 1) < MAXIMUM\_CAPACITY &&

oldCap >= DEFAULT\_INITIAL\_CAPACITY)

newThr = oldThr << 1; // double threshold

}

```

#### 功能说明：

- 如果当前容量已经等于最大限制（`MAXIMUM\_CAPACITY`），则不再扩容。

- 否则将新容量设为原来的两倍（`oldCap << 1`）。

- 如果新容量未超过最大值，并且原容量 ≥ 默认初始容量（16），则新阈值也翻倍。

> ✅ 这符合你在记忆知识中提到的“扩容策略为增加 100%”。

---

### 2. 如果当前未初始化，但已有阈值（由构造方法传入）

```java

else if (oldThr > 0) // initial capacity was placed in threshold

newCap = oldThr;

```

#### 功能说明：

- 构造时若指定了初始容量，则在首次插入时使用该容量作为新数组大小。

---

### 3. 如果既没有容量也没有阈值（默认初始化）

```java

else { // zero initial threshold signifies using defaults

newCap = DEFAULT\_INITIAL\_CAPACITY;

newThr = (int)(DEFAULT\_LOAD\_FACTOR \* DEFAULT\_INITIAL\_CAPACITY);

}

```

#### 功能说明：

- 使用默认初始容量（16）和负载因子（0.75）计算新容量和新阈值。

---

## 三、设置新阈值（如果前面没设置）

```java

if (newThr == 0) {

float ft = (float)newCap \* loadFactor;

newThr = (newCap < MAXIMUM\_CAPACITY && ft < (float)MAXIMUM\_CAPACITY ?

(int)ft : Integer.MAX\_VALUE);

}

threshold = newThr;

```

#### 功能说明：

- 如果新阈值仍未确定，则根据新容量和负载因子重新计算。

- 确保不超过整数最大值。

---

## 四、创建新数组并迁移数据

```java

@SuppressWarnings({"rawtypes","unchecked"})

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

table = newTab;

```

#### 功能说明：

- 创建一个大小为 `newCap` 的新数组。

- 将 `table` 指向新数组。

---

## 五、迁移旧数组内容到新数组

```java

if (oldTab != null) {

for (int j = 0; j < oldCap; ++j) {

Node<K,V> e;

if ((e = oldTab[j]) != null) {

oldTab[j] = null;

```

### 遍历旧数组每个桶（bucket）

#### 若当前桶只有一个节点（无冲突）

```java

if (e.next == null)

newTab[e.hash & (newCap - 1)] = e;

```

- 直接放入新数组对应位置。

- `(e.hash & (newCap - 1)` 是快速取模运算（因为数组长度始终为 2 的幂）。

---

#### 若当前桶是一个红黑树节点

```java

else if (e instanceof TreeNode)

((TreeNode<K,V>)e).split(this, newTab, j, oldCap);

```

- 调用 `split()` 方法将红黑树拆分并重新分布到新数组中。

---

#### 若当前桶是一个链表

```java

else { // preserve order

Node<K,V> loHead = null, loTail = null;

Node<K,V> hiHead = null, hiTail = null;

Node<K,V> next;

do {

next = e.next;

if ((e.hash & oldCap) == 0) {

if (loTail == null)

loHead = e;

else

loTail.next = e;

loTail = e;

} else {

if (hiTail == null)

hiHead = e;

else

hiTail.next = e;

hiTail = e;

}

} while ((e = next) != null);

```

##### 分析：

- 根据 `(e.hash & oldCap)` 是否为 0，将链表拆分为两个子链表：

- `loHead`：留在原位置的节点。

- `hiHead`：放在 `j + oldCap` 的位置（即新数组后半部分）。

- 这样可以避免重新哈希，提升性能。

```java

if (loTail != null) {

loTail.next = null;

newTab[j] = loHead;

}

if (hiTail != null) {

hiTail.next = null;

newTab[j + oldCap] = hiHead;

}

}

```

- 设置链表尾部为 `null`，结束链表。

- 将两个链表头插入新数组对应位置。

---

## 六、返回新数组

```java

return newTab;

```

---

## 七、`treeifyBin` 方法详解

```java

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);

}

}

```

### 功能说明：

- 当链表长度超过阈值（默认为 8）且数组长度 ≥ 64 时调用此方法。

- 如果数组太小（< `MIN\_TREEIFY\_CAPACITY`），则优先扩容而不是转红黑树。

- 否则，将链表节点转换为红黑树节点，并建立双向链表结构。

- 最后调用 `treeify()` 完成红黑树的平衡构建。

---

## 八、结合记忆知识验证实现规范

| 记忆知识点 | 是否满足 | 说明 |

|------------|----------|------|

| ✅ 数组长度始终为 2 的幂次方 | 是 | 通过 `resize()`

发生异常，可以输入更多信息再让我来回答或重试。