# **一、概述**

   符号表是一种用于存储键值对（key-value pair）的数据结构，我们平常经常使用的数组也可以看做是一个特殊的符号表，数组中的“键”即为数组索引，值为相应的数组元素。也就是说，当符号表中所有的键都是较小的整数时，我们可以使用数组来实现符号表，将数组的索引作为键，而索引处的数组元素即为键对应的值，但是这一表示仅限于所有的键都是比较小的整数时，否则可能会使用一个非常大的数组。散列表是对以上策略的一种“升级”，但是它可以支持任意的键而并没有对它们做过多的限定。对于基于散列表实现的符号表，若我们要在其中查找一个键，需要进行以下步骤：

* 首先我们使用**散列函数**将给定键转化为一个“数组的索引”，理想情况下，不同的key会被转为不同的索引，但在实际应用中我们会遇到不同的键转为相同的索引的情况，这种情况叫做**碰撞**。解决碰撞的方法我们后面会具体介绍。
* 得到了索引后，我们就可以像访问数组一样，通过这个索引访问到相应的键值对。

    以上就是散列表的核心思想，散列表是时空权衡的经典例子。当我们的空间无限大时，我们可以直接使用一个很大的数组来保存键值对，并用key作为数组索引，因为空间不受限，所以我们的键的取值可以无穷大，因此查找任何键都只需进行一次普通的数组访问。反过来，若对查找操作没有任何时间限制，我们就可以直接使用链表来保存所有键值对，这样把空间的使用降到了最低，但查找时只能顺序查找。在实际的应用中，我们的时间和空间都是有限的，所以我们必须在两者之间做出权衡，散列表就在时间和空间的使用上找到了一个很好的平衡点。散列表的一个优势在于我们只需调整散列算法的相应参数而无需对其他部分的代码做任何修改就能够在时间和空间的权衡上做出策略调整。

# **二、散列函数**

   介绍散列函数前，我们先来介绍几个散列表的基本概念。在散列表内部，我们使用**桶（bucket）**来保存键值对，我们前面所说的数组索引即为桶号，决定了给定的键存于散列表的哪个桶中。散列表所拥有的桶数被称为散列表的**容量（capacity）**。

   现在假设我们的散列表中有M个桶，桶号为0到M-1。我们的散列函数的功能就是把任意给定的key转为[0, M-1]上的整数。我们对散列函数有两个基本要求：一是计算时间要短，二是尽可能把键分布在不同的桶中。对于不同类型的键，我们需要使用不同的散列函数，这样才能保证有比较好的散列效果。

我们使用的散列函数应该尽可能满足均匀散列假设，以下对均匀散列假设的定义来自于Sedgewick的《算法》一书：

（均匀散列假设）我们使用的散列函数能够均匀并独立地将所有的键散布于0到M – 1之间。

    以上定义中有两个关键字，第一个是均匀，意思是我们对每个键计算而得的桶号有M个“候选值”，而均匀性要求这M个值被选中的概率是均等的；第二个关键字是独立，它的意思是，每个桶号被选中与否是相互独立的，与其他桶号是否被选中无关。这样一来，满足均匀性与独立性能够保证键值对在散列表的分布尽可能的均匀，不会出现“许多键值对被散列到同一个桶，而同时许多桶为空”的情况。

    显然，设计一个较好的满足均匀散列假设的散列函数是不容易的，好消息是通常我们无需设计它，因为我们可以直接使用一些基于概率统计的高效的实现，比如Java中许多常用的类都重写了hashCode方法（Object类的hashCode方法默认返回对象的内存地址），用于为该类型对象返回一个hashCode，通常我们用这个hashCode除以桶数M的余数就可以获取一个桶号。下面我们以Java中的一些类为例，来介绍一下针对不同数据类型的散列函数的实现。

## **1. String类的hashCode方法**

    String类的hashCode方法如下所示

[IMG_256](https://www.cnblogs.com/absfree/p/javascript:void(0);)

public int hashCode() {

int h = hash;

if (h == 0 && value.length > 0) {

char val[] = value;

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

h = 31 \* h + val[i];

}

hash = h;

}

return h;

}

[IMG_257](https://www.cnblogs.com/absfree/p/javascript:void(0);)

    hashCode方法中的value是一个char[]数组，存储中字符串的的每字符。我们可以看到在方法的最开始我们会把hash赋给h，这个hash就表示之前计算的hashCode，这样以来若之前已经计算过这个字符串对象的hashCode，这次我们就无需再计算了，直接返回之前计算过得即可。这种把hashCode缓存的策略只对不可变对象有效，因为不可变对象的hashCode是不会变的。

    根据上面的代码我们可以知道，若h为null，意味着我们是第一次计算hashCode，if语句体中就是hashCode的具体计算方法。假设我们的字符串对象str包含4个字符，ck表示的是字符串中的第k个字符（从0开始计数），那么str的hashCode就等于：31 \* (31 \* (31 \* c0 + c1) + c2) +c3。

## **2. 数值类型的hashCode方法**

    这里我们以Integer和Double为例，介绍一下数值类型的hashCode方法的一般实现。

    Integer类的hashCode方法如下：

[IMG_256](https://www.cnblogs.com/absfree/p/javascript:void(0);)

public int hashCode() {

return Integer.hashCode(value);

}

public static int hashCode(int value) {

return value;

}

[IMG_257](https://www.cnblogs.com/absfree/p/javascript:void(0);)

    其中value表示Integer对象所包装的整型值，所以Integer类的hashCode方法仅仅是简单的返回了自身的值。

    我们再来看一下Double类的hashCode方法：

[IMG_258](https://www.cnblogs.com/absfree/p/javascript:void(0);)

@Override

public int hashCode() {

return Double.hashCode(value);

}

public static int hashCode(double value) {

long bits = doubleToLongBits(value);

return (int)(bits ^ (bits >>> 32));

}

[IMG_259](https://www.cnblogs.com/absfree/p/javascript:void(0);)

    我们可以看到Double类的hashCode方法首先会将它的值转为long类型，然后返回低32位和高32位的异或的结果作为hashCode。

## **3. Date类的hashCode方法**

    前面我们介绍的数据类型都可以看做一种数值型（String可以看做一个整型数组），那么对于非数值类型对象的hashCode要怎么计算呢，这里我们以Date类为例简单的介绍一下。Date类的hashCode方法如下：

[IMG_256](https://www.cnblogs.com/absfree/p/javascript:void(0);)

public int hashCode() {

long ht = this.getTime();

return (int) ht ^ (int) (ht >> 32);

}

[IMG_257](https://www.cnblogs.com/absfree/p/javascript:void(0);)

    我们可以看到，它的hashCode方法的实现非常简单，只是返回了Date对象所封装的时间的低32位和高32位的异或结果。从Date类的hashCode的实现我们可以了解到，对于非数值类型的hashCode的计算，我们需要选取一些能区分各个类实例的实例域来作为计算的因子。比如对于Date类来说，通常具有相同的时间的Date对象我们认为它们相等，因此也就具有相同的hashCode。这里我们需要说明一下，对于等价的两个对象（也就是调用equals方法返回true），它们的hashCode必须相同，而反之则不然。

## **4. 由hashCode获取桶号**

    前面我们介绍了计算对象hashCode的一些方法，那么我们获取了hashCode之后，如何进一步得到桶号呢？一个直接的办法就是直接拿得到的hashCode除以capacity（桶的数量），然后用所得的余数作为桶号。不过在Java中，hashCode是int型的，而Java中的int型均为有符号，所以我们要是直接使用返回的hashCode的话可能会得到一个负数，显然桶号是不能为负的。所以我们先将返回的hashCode转变为一个非负整数，再用它除以capacity取余数，作为key的对应桶号，具体代码如下：

private int hash(K key) {

return (x.hashCode() & 0x7fffffff) % M;

}

    现在我们已经知道了如何通过一个键获取桶号，那么接下来我们来介绍使用散列表查找的第二步——处理碰撞。

# **三、使用拉链法处理碰撞**

    使用不同的碰撞处理方式，我们便得到了散列表的不同实现。首先我们要介绍的是使用拉链法来处理碰撞的散列表的实现。以这种方式实现的散列表，每个桶里都存放了一个链表。初始时所有链表均为空，当一个键被散列到一个桶时，这个键就成为相应桶中链表的首结点，之后若再有一个键被散列到这个桶（即发生碰撞），第二个键就会成为链表的第二个结点，以此类推。这样一来，当桶数为M，散列表中存储的键值对数目为N时，平均每个桶中的链表包含的结点数为N / M。因此，当我们查找一个键时，首先通过散列函数确定它所在的桶，这一步所需时间为O(1)；然后我们依次比较桶中结点的键与给定键，若相等则找到了指定键值对，这一步所需时间为O(N / M)。所以查找操作所需的时间为O(N / M)，而通常我们都能够保证N是M的常数倍，所以散列表的查找操作的时间复杂度为O(1)，同理我们也可以得到插入操作的复杂度也为O(1)。

    理解了以上的描述，实现基于拉链法的散列表也就很容易了，这里简单起见，我们直接使用前面的SeqSearchList作为桶中的链表，参考代码如下：

[IMG_256](https://www.cnblogs.com/absfree/p/javascript:void(0);)

public class ChainingHashMap<K, V> {

private int num; //当前散列表中的键值对总数

private int capacity; //桶数

private SeqSearchST<K, V>[] st; //链表对象数组

public ChainingHashMap(int initialCapacity) {

capacity = initialCapacity;

st = (SeqSearchST<K, V>[]) new Object[capacity];

for (int i = 0; i < capacity; i++) {

st[i] = new SeqSearchST<>();

}

}

private int hash(K key) {

return (key.hashCode() & 0x7fffffff) % capacity;

}

public V get(K key) {

return st[hash(key)].get(key);

}

public void put(K key, V value) {

st[hash(key)].put(key, value);

}

}

[IMG_257](https://www.cnblogs.com/absfree/p/javascript:void(0);)

    在上面的实现中，我们固定了散列表的桶数，当我们明确知道我们要插入的键值对数目最多只能到达桶数的常数倍时，固定桶数是完全可行的。但是若键值对数目会增长到远远大于桶数，我们就需要动态调整桶数的能力。实际上，散列表中的键值对数与桶数的比值叫做负载因子（load factor）。通常负载因子越小，我们进行查找所需时间就越短，而空间的使用就越大；若负载因子较大，则查找时间会变长，但是空间使用会减小。比如，Java标准库中的HashMap就是基于拉链法实现的散列表，它的默认负载因子为0.75。HashMap实现动态调整桶数的方式是基于公式loadFactor = maxSize / capacity，其中maxSize为支持存储的最大键值对数，而loadFactor和capacity（桶数）都会在初始化时由用户指定或是由系统赋予默认值。当HashMap中的键值对的数目达到了maxSize时，就会增大散列表中的桶数。

   以上代码中还用到了SeqSearchST，实际上这就是一个基于链表的符号表实现，支持向其中添加key-value pair，查找指定键时使用的是顺序查找，它的代码如下：

[IMG_258](https://www.cnblogs.com/absfree/p/javascript:void(0);)

public class SeqSearchST<K, V> {

private Node first;

private class Node {

K key;

V val;

Node next;

public Node(K key, V val, Node next) {

this.key = key;

this.val = val;

this.next = next;

}

}

public V get(K key) {

for (Node node = first; node != null; node = node.next) {

if (key.equals(node.key)) {

return node.val;

}

}

return null;

}

public void put(K key, V val) {

//先查找表中是否已存在相应key Node node;

for (node = first; node != null; node = node.next) {

if (key.equals(node.key)) {

node.val = val;

return;

}

}

//表中不存在相应key

first = new Node(key, val, first);

}

}

# **四、使用线性探测法处理碰撞**

## **1. 基本原理与实现**

    线性探测法是另一种散列表的实现策略的具体方法，这种策略叫做开放定址法。开放定址法的主要思想是：用大小为M的数组保存N个键值对，其中M > N，数组中的空位用于解决碰撞问题。

    线性探测法的主要思想是：当发生碰撞时（一个键被散列到一个已经有键值对的数组位置），我们会检查数组的下一个位置，这个过程被称作线性探测。线性探测可能会产生三种结果：

* 命中：该位置的键与要查找的键相同；
* 未命中：该位置为空；
* 该位置的键和被查找的键不同。

    当我们查找某个键时，首先通过散列函数得到一个数组索引后，之后我们就开始检查相应位置的键是否与给定键相同，若不同则继续查找（若到数组末尾也没找到就折回数组开头），直到找到该键或遇到一个空位置。由线性探测的过程我们可以知道，若数组已满的时候我们再向其中插入新键，会陷入无限循环之中。

    理解了以上原理，要实现基于线性探测法的散列表也就不难了。这里我们使用数组keys保存散列表中的键，数组values保存散列表中的值，两个数组同一位置上的元素共同确定一个散列表中的键值对。具体代码如下：

[IMG_256](https://www.cnblogs.com/absfree/p/javascript:void(0);)

public class LinearProbingHashMap<K, V> {

private int num; //散列表中的键值对数目

private int capacity;

private K[] keys;

private V[] values;

public LinearProbingHashMap(int capacity) {

keys = (K[]) new Object[capacity];

values = (V[]) new Object[capacity];

this.capacity = capacity;

}

private int hash(K key) {

return (key.hashCode() & 0x7fffffff) % capacity;

}

public V get(K key) {

int index = hash(key);

while (keys[index] != null && !key.equals(keys[index])) {

index = (index + 1) % capacity;

}

return values[index]; //若给定key在散列表中存在会返回相应value，否则这里返回的是null

}

public void put(K key, V value) {

int index = hash(key);

while (keys[index] != null && !key.equals(keys[index])) {

index = (index + 1) % capacity;

}

if (keys[index] == null) {

keys[index] = key;

values[index] = value;

return;

}

values[index] = value;

num++;

}  
}

[IMG_257](https://www.cnblogs.com/absfree/p/javascript:void(0);)

## **2. 动态调整数组大小**

    在我们上面的实现中，数组的大小为桶数的2倍，不支持动态调整数组大小。而在实际应用中，当负载因子（键值对数与数组大小的比值）接近1时，查找操作的时间复杂度会接近O(n)，而当负载因子为1时，根据我们上面的实现，while循环会变为一个无限循环。显然我们不想让查找操作的复杂度退化至O(n)，更不想陷入无限循环。所以有必要实现动态增长数组来保持查找操作的常数时间复杂度。当键值对总数很小时，若空间比较紧张，可以动态缩小数组，这取决于实际情况。

    要实现动态改变数组大小，只需要在上面的put方法最开始加上一个如下的判断：

if (num == capacity / 2) {

resize(2 \* capacity);

}

    resize方法的逻辑也很简单：

[IMG_258](https://www.cnblogs.com/absfree/p/javascript:void(0);)

private void resize(int newCapacity) {

LinearProbingHashMap<K, V> hashmap = new LinearProbingHashMap<>(newCapacity);

for (int i = 0; i < capacity; i++) {

if (keys[i] != null) {

hashmap.put(keys[i], values[i]);

}

}

keys = hashmap.keys;

values = hashmap.values;

capacity = hashmap.capacity;

}

[IMG_259](https://www.cnblogs.com/absfree/p/javascript:void(0);)

    关于负载因子与查找操作的性能的关系，这里贴出《算法》（Sedgewick等）中的一个结论：

在一张大小为M并含有N = a\*M（a为负载因子）个键的基于线性探测的散列表中，若散列函数满足均匀散列假设，命中和未命中的查找所需的探测次数分别为：

~ 1/2 \* (1 + 1/(1-a))和~1/2\*(1 + 1/(1-a)^2)

    关于以上结论，我们只需要知道当a约为1/2时，查找命中和未命中所需的探测次数分别为1.5次和2.5次。还有一点就是当a趋近于1时，以上结论中的估计值的精度会下降，不过我们在实际应用中不会让负载因子接近1，为了保持良好的性能，在上面的实现中我们应保持a不超过1/2。