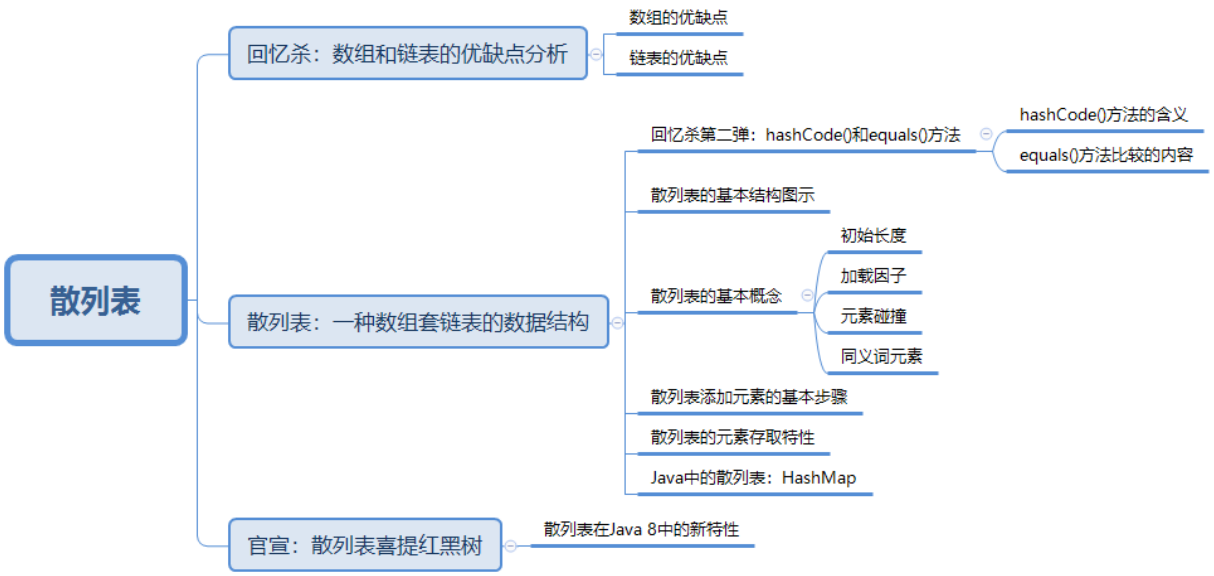


0.回忆杀：数组和链表的优缺点分析

在最开始讲解数据结构的时候，我们就首先学习了数据结构中最为重要的两种基本线性存储结构：数组和链表

并且在学习过程中我们发现：数组和链表的内存特性正好相反，互相之间能够弥补对方的不足，但是自身的缺陷却又成为了对方的优势

这样一来，我们在单纯使用一种线性存储结构的时候，必然要舍弃其中的一部分特性，从而换取另一部分优点

也就是说：

使用数组是牺牲增删效率，换取查找效率

使用链表是牺牲查找效率，换取增删效率

那么有没有一种数据结构，能够尽可能的整合二者之间的优势，规避二者之间的缺点呢？

那么答案就是散列表结构

散列表结构是由数组和链表两种结构共同组成的，所以在学习散列表结构之前

我们先来回顾一下数组和链表在内存中的特性

①数组的优缺点

数组的内存特点是：定长且连续

数组的优点：遍历快

数组在内存中，每一个元素之间的内存地址是连续的，这也就使得我们在得到数组首元素地址和被访问元素下标之后

可以直接计算得到目标元素的内存地址，而不需要一个下标一个下标的进行比对

这种访问方式能够大大提升访问数组中元素的效率，被称之为快速随机访问（Quick Random Access）

数组的缺点：增删慢

例如Java、C/C++等编程语言中实现的数组结构，其长度在内存中是不可变的

也就是说，一旦在内存空间中声明一个数组，并指定其长度，那么这个数组从诞生到被GC回收，其长度是不可变的

那么，如果我们想要向数组中插入新元素，或者删除掉数组中的某个元素，就需要我们在内存中重新声明一个数组，并且还涉及到原始元素的拷贝

这样一来，无疑是大大降低了数组在增删操作方面的效率

所以，数组定长且连续的内存特性，导致其增删慢遍历快的使用特点

②链表的优缺点

链表的内存特点是：不定长且不连续

链表的优点：增删快

链表在程序中的实现方式，是使用定义好的链表节点保存数据，并且在节点与节点之间使用引用（或者叫做指针）的方式互相关联

链表的每一个节点都相当于一个独立的对象，既然是独立的对象，那么内存地址也就是不相连的

所以在每一个节点的后面，都有一个后继节点指针，用来记录下一个节点的内存地址

所以在向链表中插入一个新节点的时候，我们并不用像数组那样，重新开辟整个存储空间，而是只要改变几个节点之间后继指针域内的内存地址即可

这样的元素节点添加方式，效率非常高

链表的缺点：遍历慢

在上面的内容中我们说到，在整个链表的存储结构中，所有的节点都相当于独立的对象，其内存地址之间是不连续的

这也就决定了，在按照下标访问链表节点的时候，我们必须从链表头节点出发，逐个遍历链表中的节点并进行下标计数

这也就导致了对链表中元素的访问速度非常慢

所以，链表不定长且不连续的内存特性，导致其增删快遍历慢的使用特点

通过上面的回忆，我们已经大致过了一遍数组和链表的特性，并且回忆了其使用特点

那么下面，就让我们一起来学习使用数组和链表共同构成的一种全新的数据结构：散列表

1.散列表：一种数组套链表的数据结构

散列表的实现，通常是使用数组和链表共同完成的。那么在一个散列表中，数组和链表之间又是什么关系呢？

答案是“数组套链表”

至于数组是如何套用链表结构的，我们会在后面的内容中进行介绍，下面我们先来看一些在学习散列表结构时需要使用到的相关概念

①回忆杀第二弹：hashCode()和equals()方法

在接触面向对象编程的时候，我们就学过：在类中最好重写hashCode()和equals()方法

通过JDK的源码我们可以知道，这两个方法都是定义在Object类当中的

也就是说，我们任何了一个类中，不论是否重写，都会具有这两个方法

那么这两个方法具体代表了什么含义，重写的时候又是怎么产生的？在类中又具有什么样的作用呢？我们一起来回顾一下

1.hashCode()方法的含义

在最终父类Object类中，我们可以看到hashCode()方法的源码是这样定义的：

```
1 public native int hashCode();
```

嗯……一个通过C/C++实现的本地方法……基本和没看是一样的……

但是我们可以通过上面的英文注释发现，这个方法就是为通过Hash结构（也就是散列结构）实现的存储结构中，存储元素时提供的方法

并且注释中还明确提到，散列结构就是类似于java.util.HashMap类型的结构

那么也就是说，这个方法一定和散列表是相关的（我觉得这就是一句废话……）

那么hashCode()方法到底返回了一个什么东西呢？

我们来运行一下下面的代码：

```
1 public class Person {
2
```

```
    private String name;
    private int age;
    private String id;

    public Person() {
    }

    public Person(String name, int age, String id) {
        this.name = name;
        this.age = age;
        this.id = id;
    }

    public static void main(String[] args) {

        Person p1 = new Person("张三", 22, "111111");
        System.out.println(p1.hashCode());

        Person p2 = new Person("李四", 23, "222222");
        System.out.println(p2.hashCode());

        Person p3 = new Person("张三", 22, "111111");
        System.out.println(p3.hashCode());

    }

}
```

上述代码的运行结果：

```
1163157884
1956725890
356573597
```

通过上述代码的运行结果与代码进行比对我们发现：不论两个对象中存储的属性取值是否相同，只要这个对象是重新new出来的

那么两个对象在使用原生的（继承自Object类中的）hashCode()方法的时候，最终打印的int型返回值都是不相同的

但是我们都知道，我们一般不会直接使用原生的hashCode()方法，而是使用重写过后的hashCode()方法

下面让我们在Person类中重写这个hashCode()方法，重写结果如下（注意：在使用Eclipse自动重写hashCode()方法的时候，会带着equals()方法一起重写

两个方法之间的关系我们会在后面介绍）：

```
@Override
public int hashCode() {
    final int prime = 31;
    int result = 1;
    result = prime * result + age;
    result = prime * result + ((id == null) ? 0 : id.hashCode());
    result = prime * result + ((name == null) ? 0 : name.hashCode());
    return result;
}
```

然后我们再去运行之前main方法中的内容：

```
1988422526
-1389389501
1988422526
```

这次我们发现，具有相同属性取值的两个分别new出来的对象之间，hashCode()方法的取值也相同了！

反过来我们通过观察重写之后hashCode()方法的代码实现可以发现：重写之后的hashCode()方法实际上和一个对象的各个对象属性的取值是有关系的

所以才会导致两个具有相同对象属性取值的对象，hashCode()方法的取值也是相同的

但是，hashCode()方法的返回值，到底代表着什么含义呢？我们可以理解为：hashCode()方法的返回值，就是对象在内存中所有对象属性的一个“特征码”

通过这个特征码，我们只能够观察到两个对象的属性取值是否相同，也就是说，如果两个对象的类型相同，并且对应的对象属性取值相同，那么这两个对象的特征码也就是相同的

那么这个特征码在散列表结构中是如何起作用的呢？这个问题我们依然会在后面进行解答

2.equals()方法的含义

同样的，在我们重写hashCode()方法的时候，Eclipse会自动帮我们重写一份equals()方法

在分析重写之后的equals()方法之前，我们先来分析一下Object类中原始equals()方法的作用

```
public boolean equals(Object obj) {
    return (this == obj);
}
```

从上述方法内容我们不难看出：原生的equals()方法的作用和==是没有区别的，只是单纯的比较两个对象之间的内存地址

也就是说，如果两个对象都是独立new出来的，那么尽管这两个对象的各个对象属性取值完全相同，他俩的内存地址之间也是不相同的

所以，任何两个独立new出来的对象之间，不论怎么调用equals()方法，都不会返回true

下面我们再来看看重写之后的equals()方法的内容：

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Person other = (Person) obj;
    if (age != other.age)
        return false;
    if (id == null) {
        if (other.id != null)
            return false;
    } else if (!id.equals(other.id))
        return false;
    if (name == null) {
        if (other.name != null)
            return false;
    } else if (!name.equals(other.name))
        return false;
    return true;
}
```

重写之后的equals()方法，不再单纯的比较两个对象的内存地址，而是去比较两个对象之间对应的对象属性取值

并且，只有在两个对象类型相同，并且所有对应的对象属性取值完全相同的情况下，这个equals()方法才会返回true

所以我们不禁联想到：重写之后的hashCode()方法是通过对象属性产生的，重写之后的equals()方法也是用来比较对象属性的

那么我们可以得到如下推断：

如果两个对象的equals()方法比较返回值为true，那么就意味着两个对象的对象属性取值都相同，那么两个对象的hashCode()方法的返回值也应该相同，反之亦然

实际上上面的推断只有一半是正确的，这两个方法之间真正的关系是：

如果两个对象的equals()方法返回true，那么两个对象的hashCode()方法返回值则相同；但是如果两个对象的hashCode()方法返回值相同，这两个对象的equals()比较有可能返回false

这是因为：在计算一个对象hashCode()方法返回值的时候，都是通过数字进行运算的，只要是数学运算，就存在着取值相同的巧合

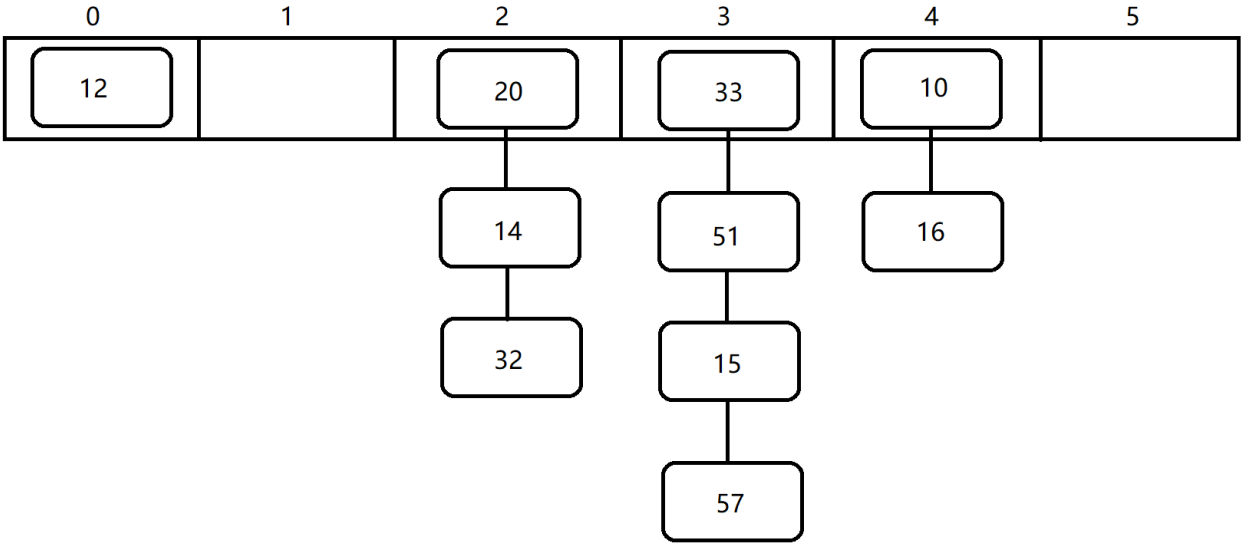
在Java中，我们利用质数（除了1和自己本身之外，不能够被其他数字整除的数）的一些数学特性，将这种可能性降到了最低

但是这种可能性还是存在的，所以我们说：即使是两个对象的hashCode()方法取值相同，也不能完全100%的保证两个对象的所有对象属性取值就一定相同

通过上面的学习，我们已经明确了hashCode()和equals()两个方法的作用和关系

那么这两个方法在散列表结构中又都是扮演着什么样的角色呢？下面就让我们来学习散列表的相关

②散列表的基本结构示意图



上图展示的，就是一个通过“数组套链表”结构实现的散列表结构，其中：

横向连续存储的是一个数组结构，数组中下挂存储的是一个个的链表结构

从图中不难看出，数组的每一个节点，保存的都是一个链表的头结点，而这些节点之间是具有一定关系性的

③散列表的基本概念

为了更好的研究散列表的特性，我们首先来说明一下散列表中的相关概念

1.初始长度

初始长度指的是整个散列表结构在初始化的时候，其数组的长度。

在初始化状态下，数组中的每一个元素所保存的都是null，也就是说在散列表初始化的时候，只具有数组结构，但是不具有链表结构

2.加载因子（Load Factor）

在我们向散列表中添加元素的时候，如果散列表的数组长度不变，那么就会导致散列表数组总有一天会被填满

此时新加入散列表的元素就只能在有限个的链表中不断下挂，这将会导致散列表的存取效率下降

所以为了避免这种情况的发生，我们设定了一个浮点值，用来表示散列表数组中，已用节点和整个数组长度的比值，即：已用元素数量 / 数组长度

如果一个正在使用的散列表结构中，数组的使用比例已经达到或者超过了这个值，那么我们将要对散列表的数组结构进行扩容

我们称这个浮点值为一个散列表的加载因子

3.元素碰撞

通过上面的图示我们能够看出来，在向散列表中添加元素的过程中，会有一些元素存放在相同的数组下标位上

那么在这样的元素加入散列表的过程中，如果发现所要保存这个元素的数组下标位上已经存在了其他元素，也就是说这个位置已经被占用了

此时我们称发生了一次元素碰撞。元素碰撞也称之为哈希碰撞

4.同义词元素

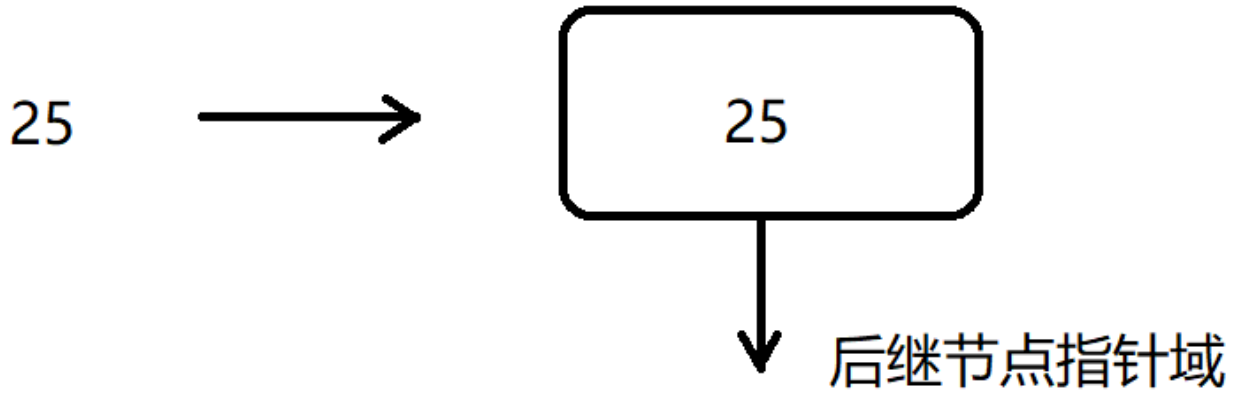
在上图中，所有保存在同一个链表结构中的元素，相互之间统称为同义词元素

④散列表添加元素的基本步骤

在了解了散列表的相关概念之后，下面我们一起来学习一下，散列表添加元素的标准步骤

步骤1：在元素加入散列表之前，散列表会创建一个节点，用于保存这个元素。

这个节点的类型可以看做是一个单链表的节点数据类型，其中除了要保存新加入的数据的取值之外，还具有一个后继节点指针域

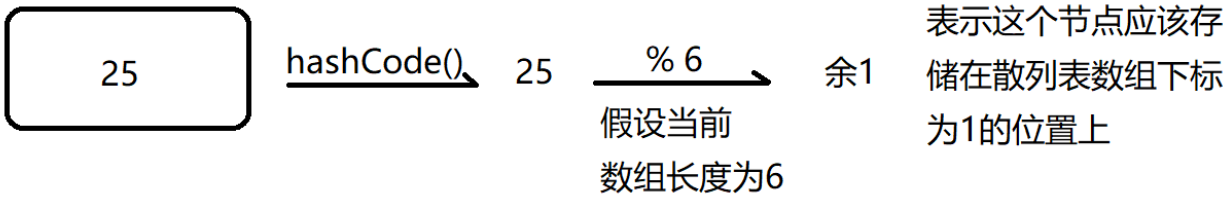


步骤2：在得到这个节点的同时，散列表还会计算加入散列表结构中元素的hashCode()编码

并通过这个元素的hashCode()编码取值，与当前散列表数组的长度进行运算，得到这个元素应该存储的数组下标位，我们称这个算法为hash算法

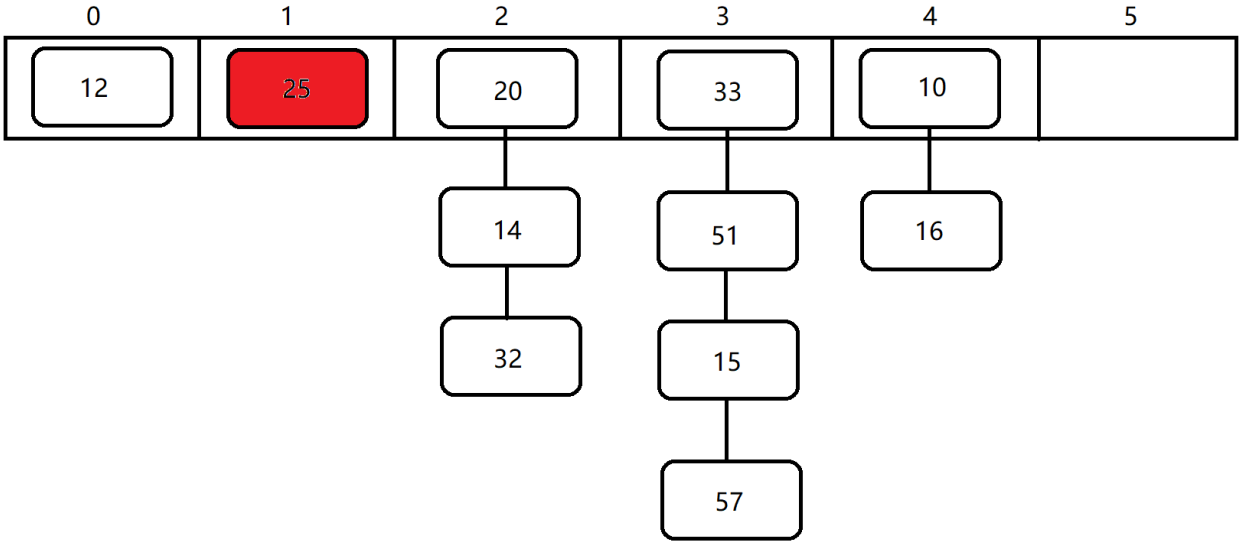
上面提到的使用hashCode()取值与数组长度之间的运算，我们现在可以简单的理解为使用对象的hashCode()编码与数组的长度进行取余运算

取得的余数，就是当前元素在散列表数组中应该存储的下标



步骤3：如果此时散列表数组中对应下标位上恰好没有任何元素，那么这个节点即直接存储在这个数组下标位上

新加入散列表的元素添加成功

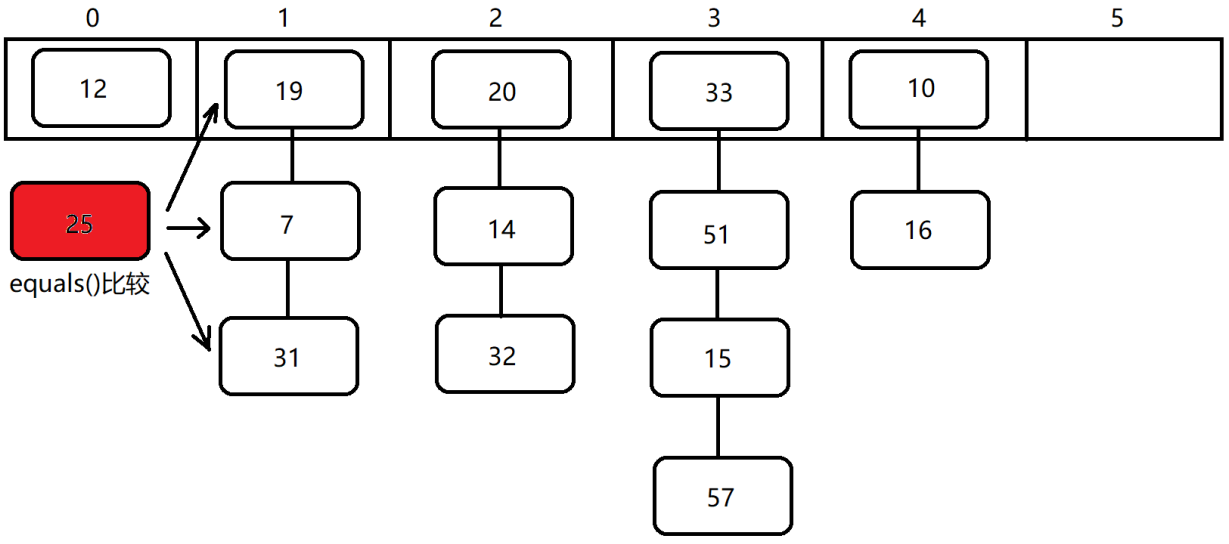


步骤4：如果当前下标位上已经存在其他节点，此时我们称发生了一次碰撞

在发证碰撞之后，我们并不会直接舍弃这个新元素的节点，而是要进行如下步骤

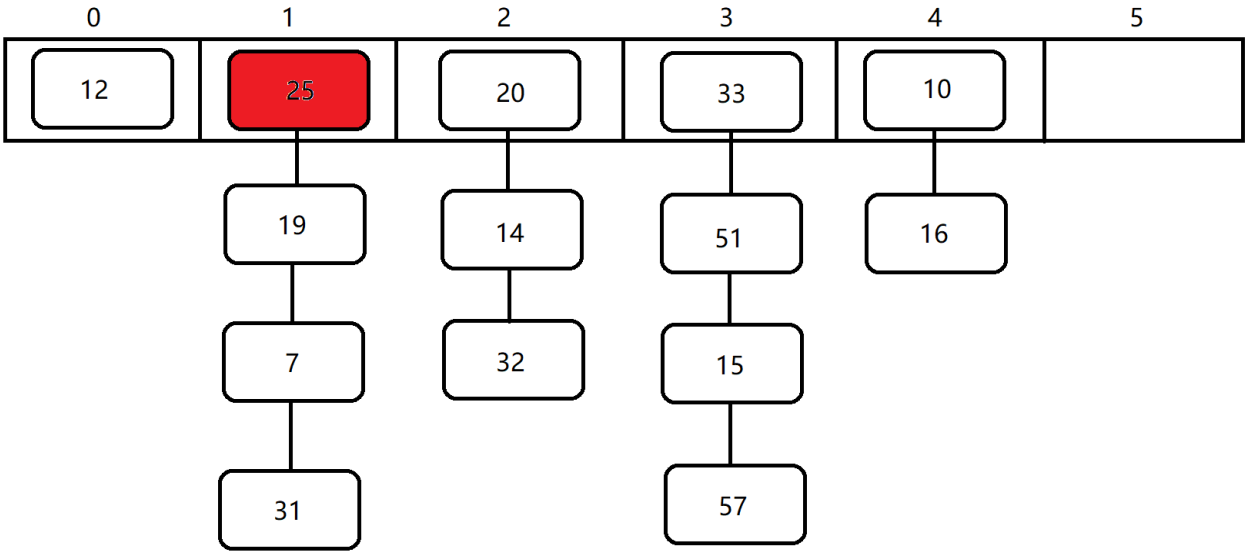
步骤5：碰撞发生后，新元素节点会沿着当前下标位上，已有元素节点构成的同义词链表进行遍历

在遍历的同时，使用equals()方法与链表中每一个节点保存的数据进行比较



步骤6：在比较过程中，如果没有发现任何元素与新加入的元素之间，equals()方法比较返回值为true，

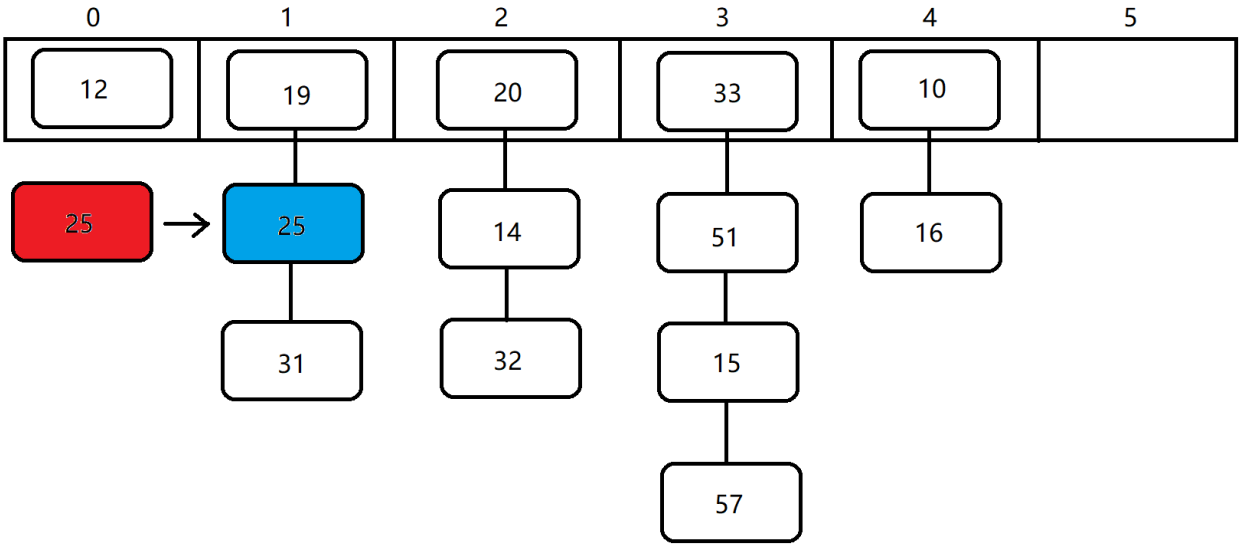
则表示在所有的同义词中，没有与新加入元素取值相同的元素，此时新加入元素的节点将会对这个同义词链表进行头插入，新元素添加成功



步骤7：如果在比较过程中，发现存在与新加入元素之间，equals()方法比较能够返回true的元素

则表示当前散列表结构中已经保存过具有相同属性取值的元素

此时放弃新加入的元素，新元素添加失败



步骤8：如果一个新元素添加成功，那么接下来散列表结构会重新计算当前数组的元素占用比例是否已经达到加载因子约定的上限

如果已经达到加载因子约定的上限，那么散列表会对数组进行扩容操作。

在数组扩容完成后，原先散列表中的元素相对于数组的下标会有一定的变化，例如：

原先当数组长度取值为6的时候，取值为25的元素所在的数组下标为1；当数组扩容为15的时候，取值为25的元素所在的数组下标变更为10

此时，散列表会对所有的元素重新分配数组存储下标，并且重新发生碰撞、比较和插入，我们称这个过程为rehash的过程

⑤散列表的元素存取特性

通过对上述散列表添加元素流程的学习我们不难发现：

1.散列表结构中不允许存在equals()方法返回true的两个元素

2.在寻找元素插入位置的时候，我们几乎都是在与数组和数组下标打交道

3.在向散列表中存储元素的时候，我们尽量少的遍历同义词链表，仅执行一次遍历，确定新元素是否能够添加进入散列表结构

　如果新元素能够被保留，则执行链表的头插入，这也是为了尽可能少的遍历链表

综合上述总结的规律，我们不难得出这样一个结论：散列表是一种平衡了数组和链表两种线性存储结构查找和增删元素方面优点的存储结构

如果单纯的讨论查询元素的效率，那么散列表不如数组查询元素快速，但是优于链表结构；

如果单纯的讨论添加和删除元素的效率，那么散列表不如链表，但是优于数组。

所以从综合表现上来说，不管是在增删元素方面，还是查找元素的方面，散列表的表现都是最居中但又是最稳定的

所以在实际开发中，我们能够用散列表的场景非常多见

⑥Java中的散列表：HashMap

在Java中，有一种很常见的散列表结构的实现类，那就是HashMap。实际上HashSet的底层是使用HashMap实现的，所以HashSet也是使用散列表实现的

但是为了观察源码方便，我们选择使用HashMap来进行说明。

在HashMap中，散列表数组的初始化长度定义为16，加载因子的取值为0.75f

但是在HashMap中存在这样的一个构造方法：

```
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;
```

```
11      this.threshold = tableSizeFor(initialCapacity);
12  }
```

通过这个方法，我们有权限自定义散列表数组的初始化长度和加载因子的取值

同样的，在HashMap中，通过对象hashCode()编码取值计算对应数组存储下标位的hash算法，也比单纯的取余数要复杂的多

2.官宣：散列表喜提红黑树

散列表在Java 8中的新特性

在Java 8中，Java开发人员对散列表的构成进行了修改

在之前的内容中我们已经学习了红黑树结构，我们知道红黑树结构是一种在元素查找和元素增删方面效率比较平衡的二叉排序树结构

所以，在Java 8中，散列表结构使用的不再是单纯的数组+链表的实现方式，而是使用数组+链表或者数组+红黑树的方式进行元素存储

使用数组+红黑树构建散列表的条件是：如果整个散列表中，元素的总数量大于等于64个，并且其中某一条链表中同义词的数量大于等于8

那么就将这条链表结构重新构建成为红黑树结构。

这种修改实际上是牺牲了一部分向散列表中添加元素的效率，换取了在散列表中，多同义词部分下查找元素的效率

其示意图如下：