

# Collection Framework

YAO ZHAO

# Overview



Collection



# 常用的集合类实现的接口及底层数据结构

Interface	Hash Table	Resizable Array	Balanced Tree	Linked List	Hash Table + Linked List
Set	<u>HashSet</u>		<u>TreeSet</u>		<u>LinkedHashSet</u>
List		<u>ArrayList</u>		<u>LinkedList</u>	
Deque		<u>ArrayDeque</u>		<u>LinkedList</u>	
Map	<u>HashMap</u>		<u>TreeMap</u>		<u>LinkedHashMap</u>

# 接口提供的功能

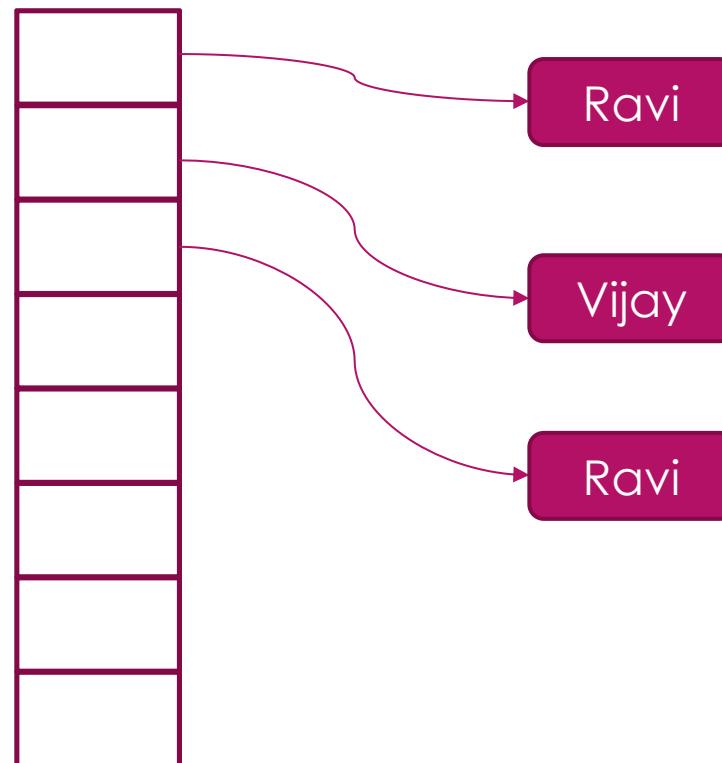
- ▶ List 有序的存储元素
- ▶ Set 不保存重复元素，无次序

有序的意思是，增加元素的时候，一般是往最后的位置添加，如果要从中间添加，该位置后面的元素都将往后挪；删除的时候，后面所有的元素都会往前挪。迭代的时候可以从前往后顺次迭代，迭代的过程中不会遇到空的存储空间。

无序的意思是，不保证元素的次序，其位置由其key值决定。无法指定位置添加值。也无法由key值直接定位，需要查找才能知道具体元素在哪。

# DemoList: ArrayList

```
list.add("Ravi");
list.add("Vijay");
list.add("Ravi");
```



# Debug and watch

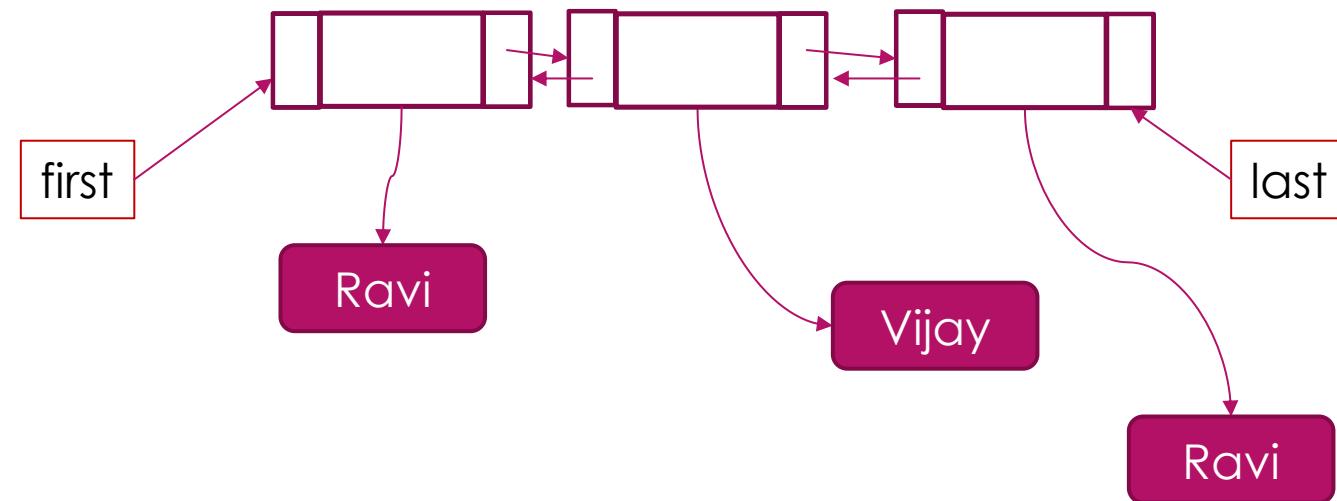
▼ ⓘ list	ArrayList<E> (id=17)
▼ ⓘ elementData	Object[10] (id=30)
► ⓘ [0]	"Ravi" (id=31)
► ⓘ [1]	null
► ⓘ [2]	null
► ⓘ [3]	null
► ⓘ [4]	null
► ⓘ [5]	null
► ⓘ [6]	null
► ⓘ [7]	null
...	"

▼ ⓘ list	ArrayList<E> (id=17)
▼ ⓘ elementData	Object[10] (id=30)
► ⓘ [0]	"Ravi" (id=31)
► ⓘ [1]	"Vijay" (id=34)
► ⓘ [2]	null
► ⓘ [3]	null
► ⓘ [4]	null
► ⓘ [5]	null
► ⓘ [6]	null
► ⓘ [7]	null
...	"

▼ ⓘ list	ArrayList<E> (id=17)
▼ ⓘ elementData	Object[10] (id=30)
► ⓘ [0]	"Ravi" (id=31)
► ⓘ [1]	"Vijay" (id=34)
► ⓘ [2]	"Ravi" (id=31)
► ⓘ [3]	null
► ⓘ [4]	null
► ⓘ [5]	null
► ⓘ [6]	null
► ⓘ [7]	null

# DemoList: LinkedList

```
list.add("Ravi");  
list.add("Vijay");  
list.add("Ravi");
```



# Debug and watch

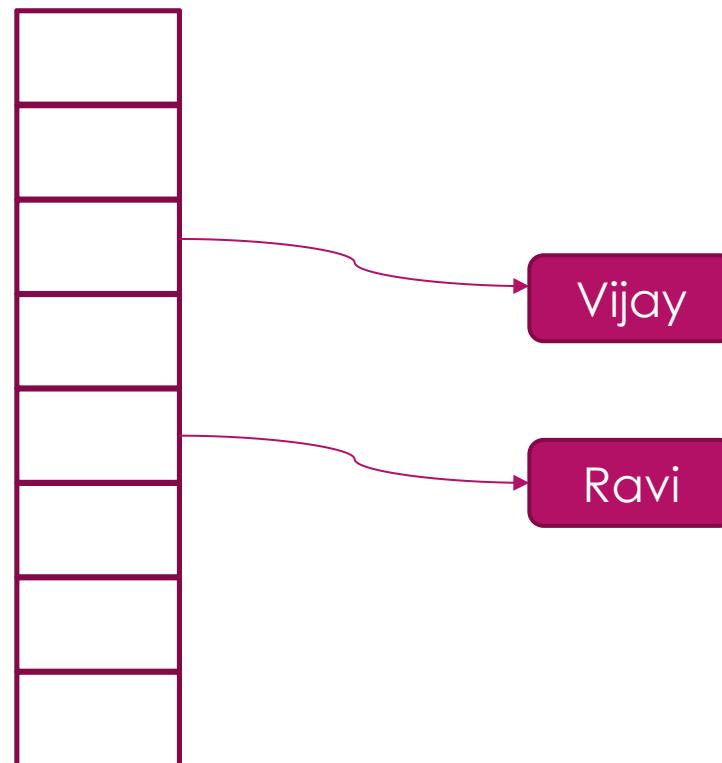
▼ ⚡ list	LinkedList<E> (id=18)
▼ ⚡ first	LinkedList\$Node<E> (id=30)
► ⚡ item	"Ravi" (id=32)
▲ next	null
▲ prev	null
► ⚡ last	LinkedList\$Node<E> (id=30)
● modCount	1
▲ size	1

▼ ⚡ list	LinkedList<E> (id=18)
▼ ⚡ first	LinkedList\$Node<E> (id=30)
▼ ⚡ item	"Ravi" (id=32)
■ hash	0
► ⚡ value	(id=37)
▼ ⚡ next	LinkedList\$Node<E> (id=35)
► ⚡ item	"Vijay" (id=36)
▲ next	null
► ⚡ prev	LinkedList\$Node<E> (id=30)
▲ prev	null
► ⚡ last	LinkedList\$Node<E> (id=35)
● modCount	2
● size	2

▼ ⚡ list	LinkedList<E> (id=18)
▼ ⚡ first	LinkedList\$Node<E> (id=30)
▼ ⚡ item	"Ravi" (id=32)
■ hash	0
► ⚡ value	(id=37)
▼ ⚡ next	LinkedList\$Node<E> (id=35)
► ⚡ item	"Vijay" (id=36)
▼ ⚡ next	LinkedList\$Node<E> (id=39)
► ⚡ item	"Ravi" (id=32)
▲ next	null
► ⚡ prev	LinkedList\$Node<E> (id=35)
► ⚡ prev	LinkedList\$Node<E> (id=30)
▲ prev	null
► ⚡ last	LinkedList\$Node<E> (id=39)

# DemoSet: HashSet

```
set.add("Ravi");  
set.add("Vijay");  
set.add("Ravi");
```



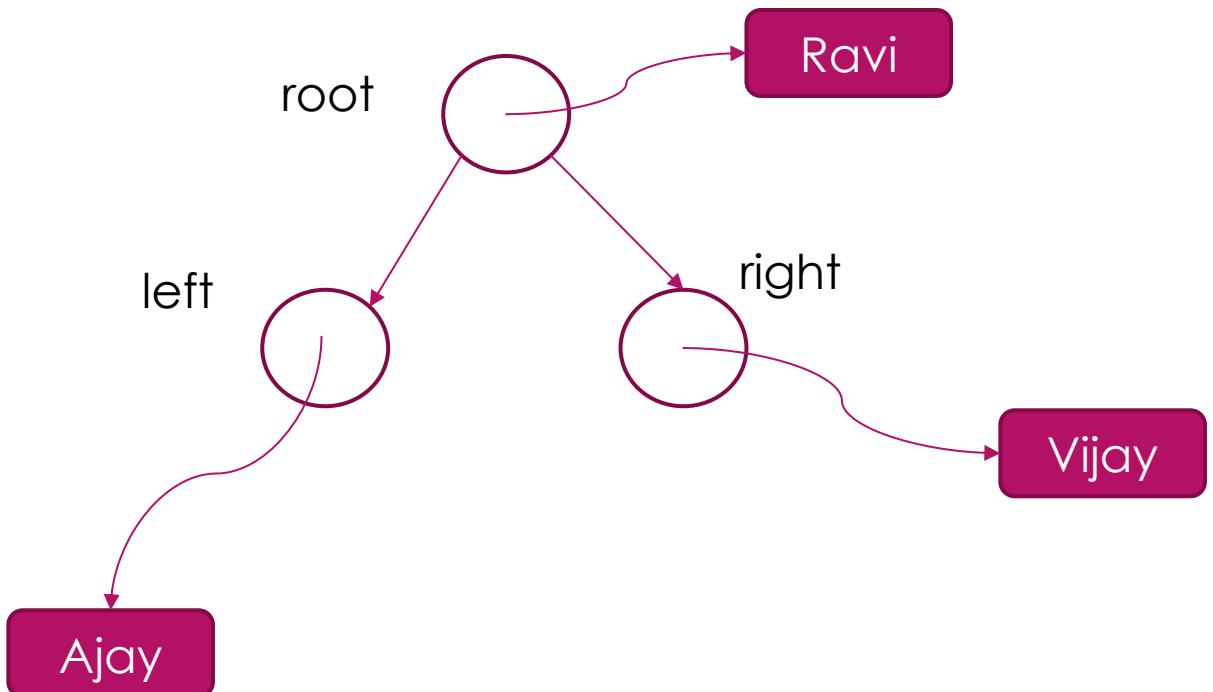
# Debug and watch

▲ table	HashMap\$Node<K,V>[16] (id=31)
▲ [0]	null
▲ [1]	null
▼ ▲ [2]	HashMap\$Node<K,V> (id=44)
▲ hash	82656898
► ▲ key	"Vijay" (id=45)
▲ next	null
▲ value	Object (id=39)
▲ [3]	null
▼ ▲ [4]	HashMap\$Node<K,V> (id=34)
▲ hash	2539876
► ▲ key	"Ravi" (id=36)
▲ next	null
▲ value	Object (id=39)
▲ [5]	null

▼ ▲ table	HashMap\$Node<K,V>[16] (id=31)
▲ [0]	null
▲ [1]	null
▲ [2]	null
▲ [3]	null
▼ ▲ [4]	HashMap\$Node<K,V> (id=34)
▲ hash	2539876
► ▲ key	"Ravi" (id=36)
▲ next	null
▲ value	Object (id=39)
▲ [5]	null
▲ [6]	null
▲ [7]	null
▲ [8]	null
▲ [9]	null

# DemoSet: TreeSet

```
set.add("Ravi");
set.add("Vijay");
set.add("Ravi");
set.add("Ajay");
```



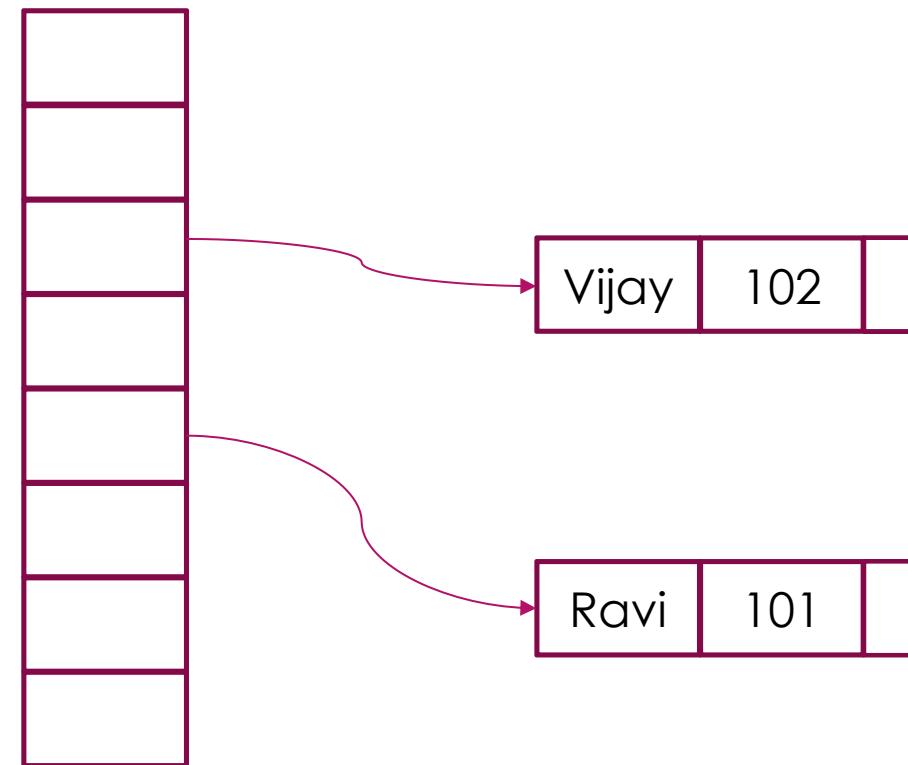
set	TreeSet<E> (id=15)
m	TreeMap<K,V> (id=29)
comparator	null
descendingMap	null
entrySet	null
keySet	null
modCount	1
navigableKeySet	null
root	TreeMap\$Entry<K,V> (id=35)
color	true
key	"Ravi" (id=38)
left	null
parent	null
right	null
value	Object (id=41)
size	1
values	null

set	TreeSet<E> (id=15)
m	TreeMap<K,V> (id=29)
comparator	null
descendingMap	null
entrySet	null
keySet	null
modCount	2
navigableKeySet	null
root	TreeMap\$Entry<K,V> (id=35)
color	true
key	"Ravi" (id=38)
left	null
parent	null
right	TreeMap\$Entry<K,V> (id=42)
color	false
key	"Vijay" (id=43)
left	null
parent	TreeMap\$Entry<K,V> (id=35)
right	null
value	Object (id=41)
value	Object (id=41)
size	2

set	TreeSet<E> (id=15)
m	TreeMap<K,V> (id=29)
comparator	null
descendingMap	null
entrySet	null
keySet	null
modCount	3
navigableKeySet	null
root	TreeMap\$Entry<K,V> (id=35)
color	true
key	"Ravi" (id=38)
left	TreeMap\$Entry<K,V> (id=47)
color	false
key	"Ajay" (id=48)
left	null
parent	TreeMap\$Entry<K,V> (id=35)
right	null
value	Object (id=41)
parent	null
right	TreeMap\$Entry<K,V> (id=42)
color	false
key	"Vijay" (id=43)
left	null
parent	TreeMap\$Entry<K,V> (id=35)
right	null
value	Object (id=41)
value	Object (id=41)
size	3

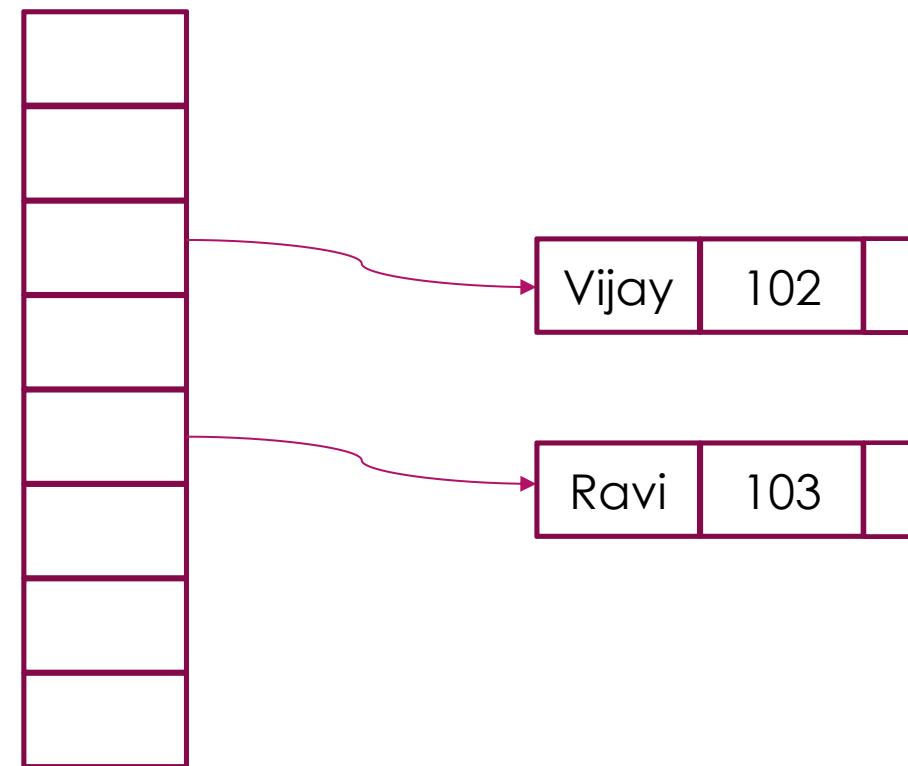
# DemoMap: HashMap

```
map.put("Ravi", 101);
map.put("Vijay", 102);
map.put("Ravi", 103);
map.put("Ajay", 104);
```



# DemoMap: HashMap

```
map.put("Ravi", 101);
map.put("Vijay", 102);
map.put("Ravi", 103);
map.put("Ajay", 104);
```

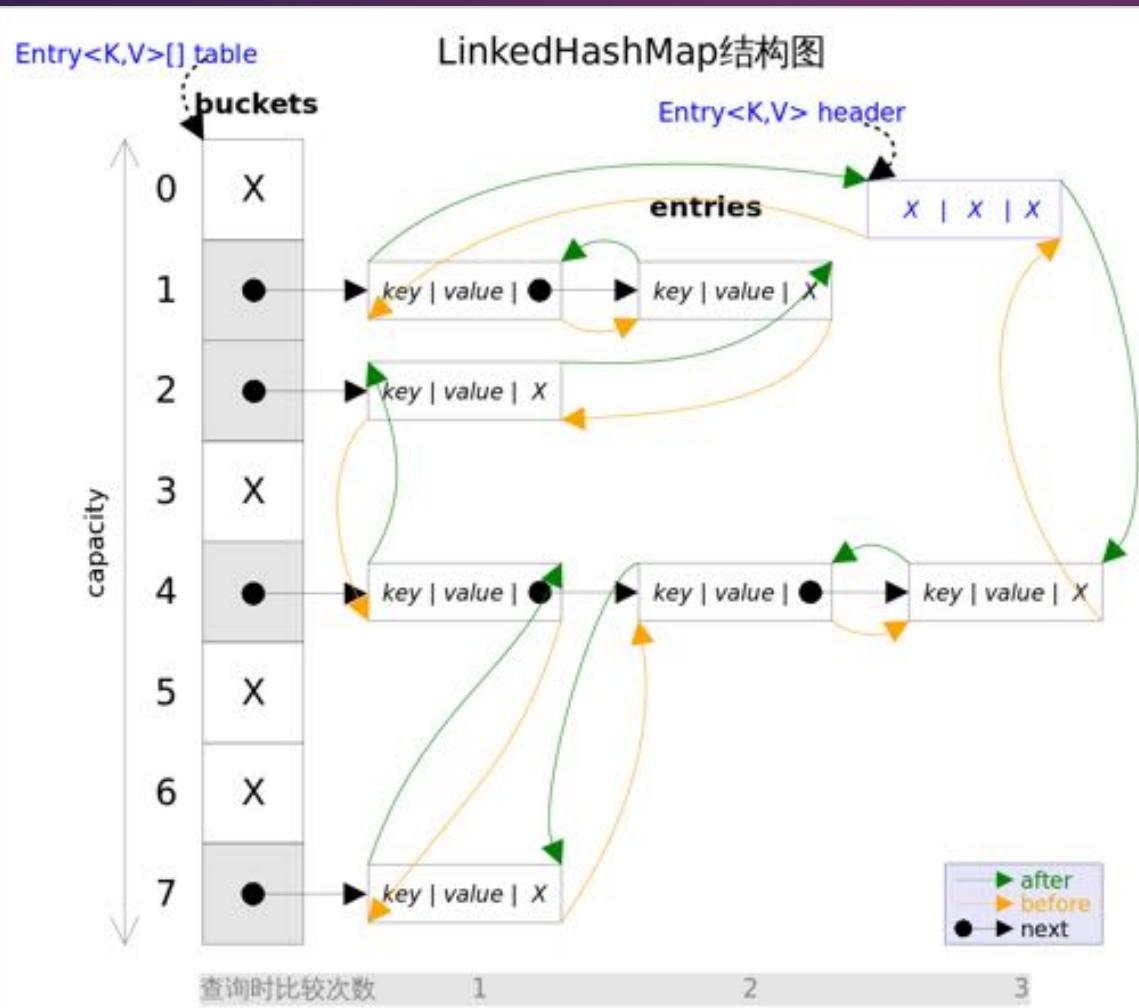


map	HashMap<K,V> (id=16)
entrySet	null
keySet	null
loadFactor	0.75
modCount	1
size	1
table	HashMap\$Node<K,V>[16] (id=24)
[0]	null
[1]	null
[2]	null
[3]	null
[4]	HashMap\$Node<K,V> (id=27) hash: 2539876 key: "Ravi" (id=29) next: null value: Integer (id=32)
value	101
[5]	null
[6]	null
[7]	null
[8]	null

map	HashMap<K,V> (id=16)
entrySet	null
keySet	null
loadFactor	0.75
modCount	2
size	2
table	HashMap\$Node<K,V>[16] (id=24)
[0]	null
[1]	null
[2]	HashMap\$Node<K,V> (id=37) hash: 82656898 key: "Vijay" (id=38) next: null value: Integer (id=39)
value	102
[3]	null
[4]	HashMap\$Node<K,V> (id=27) hash: 2539876 key: "Ravi" (id=29) next: null value: Integer (id=32)
value	101
[5]	null
[6]	null

① map	HashMap<K,V> (id=16)	
▲ entrySet	null	
▲ keySet	null	
▲ F loadFactor	0.75	
▲ modCount	2	
▲ size	2	
▼ ▲ table	HashMap\$Node<K,V>[16] (id=24)	
▲ [0]	null	
▲ [1]	null	
▼ ▲ [2]	HashMap\$Node<K,V> (id=37)	
▲ F hash	82656898	
► ▲ F key	"Vijay" (id=38)	
▲ next	null	
▼ ▲ value	Integer (id=39)	
■ F value	102	
▲ [3]	null	
▼ ▲ [4]	HashMap\$Node<K,V> (id=27)	
▲ F hash	2539876	
► ▲ F key	"Ravi" (id=29)	
▲ next	null	
▼ ▲ value	Integer (id=40)	
■ F value	103	
▲ [5]	null	
▲ [6]	null	
▲ [7]	null	
▲ [3]		
▼ ▲ [4]		
▲ [5]		
▼ ▲ [6]		
▲ [7]		
▼ ▲ [8]		
▲ [9]		
▲ [10]		
▲ [11]		
▲ [12]		
▲ [13]		
▼ ▲ [14]		
▲ F hash	HashMap\$Node<K,V> (id=41)	
► ▲ F key	"Ajay" (id=42)	
■ hash	2041438	
► ■ value	2041409	
▲ next	(id=44)	
▼ ▲ value	null	
■ F value	Integer (id=43)	
▲ [15]	104	
▲ [16]	null	

# LinkedHashMap



# 关键评估要素

- ▶ 接口决定了其功能
- ▶ 底层实现决定了其效率



需要干什么？  
哪种操作比较多？

# List

- ▶ `ArrayList`: 由数组实现的List。允许对元素进行快速随机访问，但是向List中间插入与移除元素的速度很慢。
- ▶ `LinkedList` : 向List中间插入与删除的开销并不大。随机访问则相对较慢。`LinkedList`可以当作堆栈、队列和双向队列使用。

# Set

- ▶ **Set** : 存入Set的每个元素都必须是唯一的，因为Set不保存重复元素。加入Set的元素必须定义equals()方法以确保对象的唯一性。Set与Collection的接口差不多（没有removeIf和stream）。Set接口不保证维护元素的次序。
- ▶ **HashSet**: 为快速查找设计的Set。存入HashSet的对象必须定义hashCode()。
- ▶ **TreeSet**: 保存次序的Set, 底层为树结构，使用它可以从Set中提取有序的序列。默认按照自然序排序，或者按照Comparator中实现的compare排序
- ▶ **LinkedHashSet**: 具有HashSet的查询速度，且内部使用链表维护元素的顺序(插入的次序)。于是在使用迭代器遍历Set时，结果会按元素插入的次序显示。

# Map

- ▶ Map : 维护 “K->V” 的关联性，通过 “K” 查找 “V” 。
- ▶ HashMap: 基于HashCode实现高效地插入和查询。可以通过构造器设置容量 capacity和负载因子load factor，以调整容器的性能。
- ▶ LinkedHashMap: 插入和查询类似于HashMap，但是迭代遍历时，取得“键值对”的顺序是其插入次序，或者是最近最少使用(LRU)的次序。只比HashMap慢一点。而在迭代访问时比较快(Linked的作用)。
- ▶ TreeMap : 基于红黑树实现。元素根据Key排序(次序由Comparable或 Comparator决定)。迭代获取的结果是按照Key排序的。TreeMap是唯一的带有 subMap()方法的Map，可以返回一个子树。

# 简单的场景

- ▶ 如果有10个学生，他们的学号分别为1, 2, 3.....10，老师想存储他们的年龄，要怎么存？
- ▶ 如果有10个学生，他们的学号分别为16110001, 16110002, 16110003.....16110010，老师想存储他们的年龄，要怎么存？
- ▶ 如果有10个学生，他们的名字分别为“方一”，“倪二”“张三”，“李四”，“王五”，.....“杜十”，“陈十一”，老师想存储他们的年龄，要怎么存？
- ▶ 假设某公司有4000多个教职工，教职工的工号以0开头的表示为女，以1开头的表示为男，每新报到一个教职工，就在当前最大的编号+1作为新编号。当前员工编号已经到达10000。
- ▶ 假设老师需要组建一个团队，分别包含一个生物系、计算机系、物理系、材料系的学生，通知发出后，很多学生报名，老师按照报名先后分别将需要的专业的学生加入团队。此团队可以什么类型表示，报名的学生可以什么类型表示。

# 场景分析

- ▶ 如果有10个学生，他们的学号分别为1, 2, 3.....10，老师想存储他们的年龄，并且想快速查找，要怎么存？

用数组或者ArrayList就可以

- ▶ 如果有10个学生，他们的学号分别为18110001, 18110002, 18110003.....18110010，老师想存储他们的年龄，要怎么存？

用数组或者ArrayList也可以，自己写个映射（学号-18110000），从学号映射到数组下标。

- ▶ 如果有10个学生，他们的名字分别为“方一”，“倪二”“张三”，“李四”，“王五”，.....“杜十”，“陈十一”，老师想存储他们的年龄，要怎么存？

HashMap，设置HashMap的Key为String，Value为Integer

# 场景分析

- ▶ 假设某公司有4000多个教职工，教职工的工号以0开头的表示为女，以1开头的表示为男，每新报到一个教职工，就在当前最大的编号+1作为新编号。当前员工编号已经到达10000。

如果前离职员工的信息需要保存，则用ArrayList；

如果离职员工的信息不需要保持，则用HashMap。（教职工信息使用场景，并不需要时时保持有序的状态，但是教职工信息查询是频繁的操作）

- ▶ 假设老师需要组建一个团队，分别包含一个生物系、计算机系、物理系、材料系的学生，通知发出后，很多学生报名，也有同学报名后取消报名，老师按照报名先后分别将需要的专业的学生加入团队。此团队可以什么类型表示，报名的学生可以什么类型表示。

此团队可以用HashSet表示，重写equals；

报名的学生可以用Queue，按照报名先后插入队列，取消报名则从队列中删除。

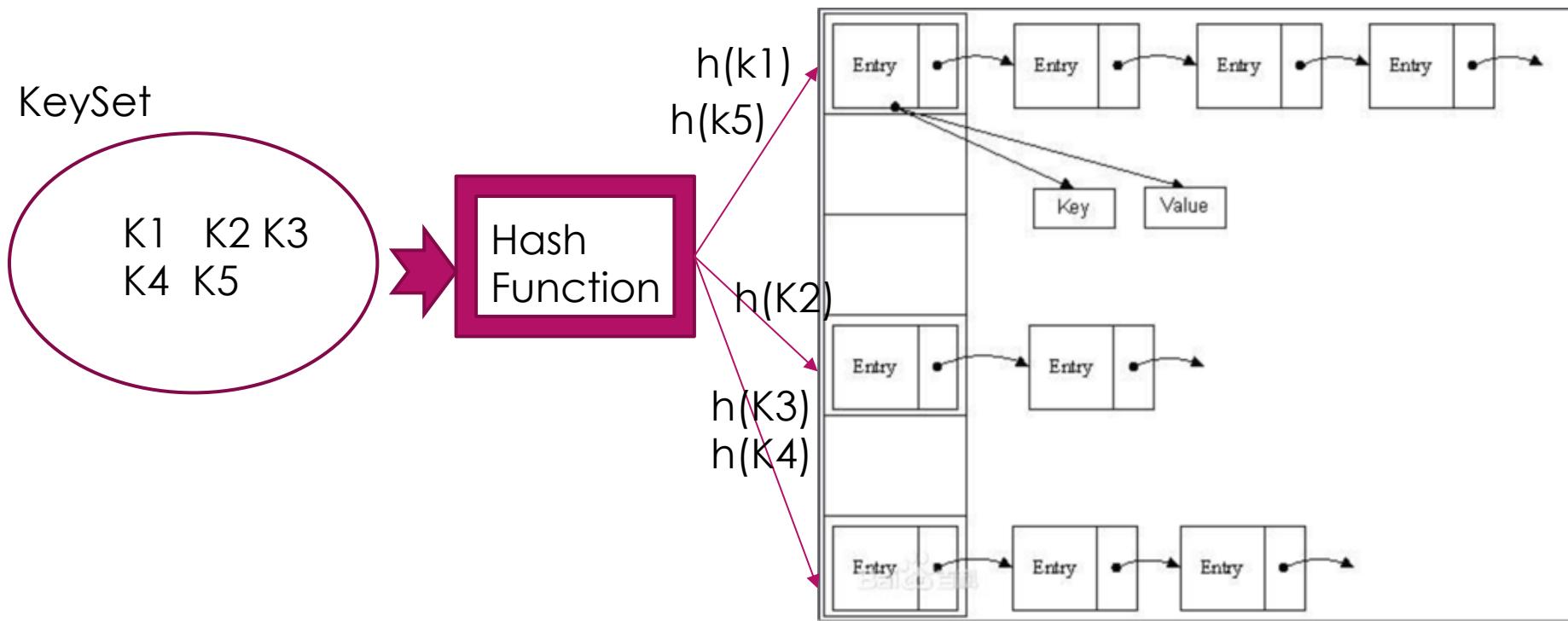
# 为什么HashMap会比较快的查找？

- ▶ Hash过程, Key映射到table的位置
- 根据key得到hashcode 方法: int hash = hash(key.hashCode());
- 根据hashcode得到对应的位置 :(n-1)&hash;
- ▶ 会不会有不同的key映射到相同的位置? 怎么解决?

# Hash函数的作用及设计要点

- ▶ 作用：哈希通过哈希函数将要检索的项与索引（散列，散列值）关联起来，生成一种便于搜索的数据结构（哈希表）
- ▶ 设计要点：
  - 哈希函数的计算简单，快速；
  - 哈希函数能将关键字集合K均匀地分布在地址集{0,1, ..., m-1}上，使冲突最小。
- ▶ 思考：一个好的Hash函数可以完全避免冲突吗？如果发生冲突，应该怎么处理？

# HashMap的结构



# Pay attention: hashCode 和equals

- ▶ Object.hashCode()的一致性原则：
  - 在程序的一次执行过程中，对同一个对象必须一致地返回同一个整数。
  - 如果两个对象通过equals(Object)比较，结果相等，那么对这两个对象分别调用 hashCode方法应该产生相同的整数结果。
  - 如果两个对象通过equals比较，结果不相等，不必保证对这两个对象分别调用 hashCode也返回两个不相同的整数。
- ▶ 默认情况下， Object.hashCode()是将该对象的内部地址转换成一个整数
- ▶ 默认情况下， Object.equals() **return this == obj;**

- ▶ 在多种包含容器操作的语言中，`hashCode`方法的主要作用是为了配合基于hash的集合一起正常运行，这样的hash集合包括`HashSet`、`HashMap`以及`HashTable`。
- ▶ 为什么有`equals`还需要`hashcode`? 效率。

# 重写hashCode注意事项

- ▶ 对同一个对象调用 hashCode() 都应该产生同样的值。如果在讲一个对象用 put() 添加进HashMap时产生一个hashCdoe值，而用get()取出时却产生了另一个 hashCode值，那么就无法获取该对象了。
- ▶ 如果重写hashCode方法将依赖于对象中可能会被修改的数据，此种操作危险，因为数据发生变化时， hashCode()方法就会生成一个不同的hashCdoe值。

# 不可变类

- ▶ 类添加final修饰符，保证类不被继承。
- ▶ 保证所有成员变量必须私有，并加final修饰符
- ▶ 不提供改变成员变量的方法，包括setter
- ▶ 通过构造器的入参初始化成员变量时，如果入参有对象或数组进行深拷贝(deep copy)
- ▶ 在getter方法中，不要直接返回对象本身，而是克隆对象，并返回对象的拷贝

- ▶ 怎样减少冲突
- ▶ 兼顾迭代的性能

# 课后推荐阅读

- ▶ 《Java核心技术 卷1 基础知识》
  - 第12章 泛型程序设计
  - 第13章 集合
- ▶ 《Java编程思想》
  - 第8章 对象的容纳