集合框架3

# 一 内容回顾（列举前一天重点难点内容）

## 1.1 教学重点:

```
1.熟练使用ArrayList和LinkedList
2.掌握数据结构分类
3.掌握常用数据结构的基本原理(数组,链表,栈,队列)
4.掌握泛型的作用
5.掌握泛型的基本使用(类上,方法上,接口上)
6.掌握工具类Collections的基本使用
```

## 1.2 教学难点:

```
1.数据结构进阶学习
数据结构很重要,是小伙伴们进行软件开发的一项必备技能,但是数据结构涉及到的知识点很多,其中还有很多逻辑性的知识,如果想一次全都掌握了,很难.对于我们要有一个学习的过程,从简单数据结构入手,再结合数据结构的实际使用,先掌握它的基本使用,再从浅入深,学习复杂数据结构.当前阶段只要求掌握数据结构分类以及常用数据结构的基本原理(数组,链表,栈,队列)即可.
2.泛型中限制下限限制上限
限制上限下限是泛型的综合使用,理解起来会有些困难,这里不做强制要求,有精力的小伙伴可以学习下.
```

# 二 教学目标

```
1.掌握HashSet的基本使用
2.掌握TreeSet的基本使用
3.掌握Comparable和Comparator的使用
4.掌握Map集合的基本方法
5.掌握Map的遍历(keySet()和entrySet())
6.掌握哈希表的基本原理
7.掌握二叉树的遍历
8.了解数据结构-哈希表实现
9.了解数据结构-二叉树的原理及实现
10.了解HashMap的去重和排序
11.了解TreeMap的去重和排序
```

# 三 教学导读

## 3.1. Set集合

- Set集合中，没有下标的概念。
- Set集合，是一个去重复的集合。 在Set集合中不会添加重复的元素的!

> 在向一个Set集合中添加元素的时候， 会先判断这个元素是否已经存在了。 如果存在， 则不再添加。

- Set集合中， 数据的存储是无序的。

> 无序： 所谓的无序， 其实指的是元素的添加顺序和存储顺序是不一致的。
>
> 无序， 并不意味着随机!

Set接口， 是继承自Collection接口的。 Set接口中的方法， 都是从Collection接口中继承下来的， 并没有添加新的方法。

## 3.2. Map集合

Map是双列集合的顶级接口， 这个接口并没有继承自Collection接口。

在Map中， 更多强调的是一层映射关系。 在Map中存储的数据， 是一个个的键值对（Key-Value-Pair）， 键和值是一一对应的。

> 需要注意:
>
> 由于Map集合并没有实现Iterable接口， 因此这个集合是不能使用增强for循环遍历的。

# 四 教学内容

## 4.1 Set集合(会)

### 4.1.1. HashSet与TreeSet的区别

- HashSet:底层是哈希表,线程不安全的
- TreeSet:底层是二叉树,线程不安全的

### 4.1.2. 哈希表(了解)

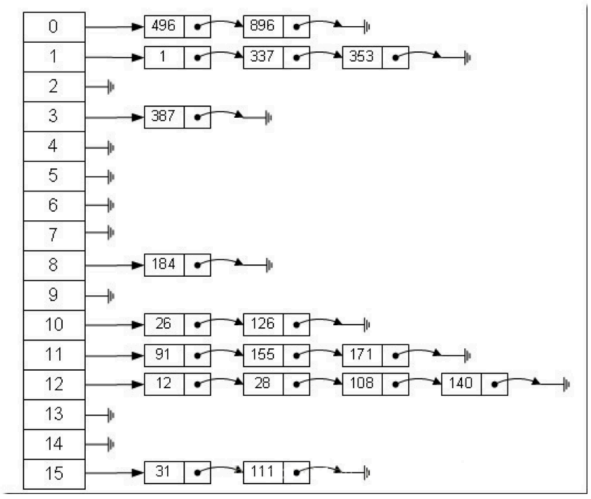
#### 4.1.2.1. 哈希表简介

Set集合的两个实现类HashSet与LinkedHashSet，底层实现都是哈希表。

- Hash，一般翻译做“散列”，也有直接音译为“哈希”的，它是基于快速存取的角度设计的，也是一种典型的“空间换时间”的做法。顾名思义，该数据结构可以理解为一个线性表，但是其中的元素不是紧密排列的，而是可能存在空隙。
- 散列表（Hash table，也叫哈希表），是根据键值码值(Key value)而直

接进行访问的数据结构。也就是说，它通过把键值码值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数。

- Hash表的组成是”数组+链表”这些元素是按照什么样的规则存储到数组中呢。一般情况是通过hash(key)%len获得，也就是元素的key的哈希值对数组长度取模得到。

> 比如下图哈希表中，
> 12%16=12,28%16=12,108%16=12,140%16=12。所以12、28、108以及140都存储在数组下标为12的位置
>
> 
> 

#### 4.1.2.2. hash表扩容的理解

可是当哈希表接近装满时,因为数组的扩容问题,性能较低(转移到更大的哈希表中).

Java默认的散列单元大小全部都是2的幂，初始值为16（2的4次幂）。假如16条链表中的75%链接有数据的时候，则认为加载因子达到默认的0.75。HahSet开始重新散列，也就是将原来的散列结构全部抛弃，重新开辟一个散列单元大小为32（2的5次幂）的散列结果，并重新计算各个数据的存储位置。以此类推下去.....

负载(加载)因子:0.75.-->hash表提供的空间是16 也就是说当到达12的时候就扩容

#### 4.1.2.3. 排重机制的实现

假如我们有一个数据(散列码76268)，而此时的HashSet有128个散列单元，那么这个数据将有可能插入到数组的第108个链表中(76268%128=108)。但这只是有可能，如果在第108号链表中发现有一个老数据与新数据equals()=true的话，这个新数据将被视为已经加入，而不再重复丢入链表。

#### 4.1.2.4. 优点

哈希表的插入和查找是很优秀的.

对于查找:直接根据数据的散列码和散列表的数组大小计算除余后，就得到了所在数组的位置，然后再查找链表中是否有这个数据即可。因为数组本身查找速度快,所以查找的效率高低体现在链表中，但是真实情况下在一条链表中的数据又很少，有的甚至没有,所以几乎没有什么迭代的代价。所以散列表的查找效率建立在散列单元所指向的链表中数据的多少上.

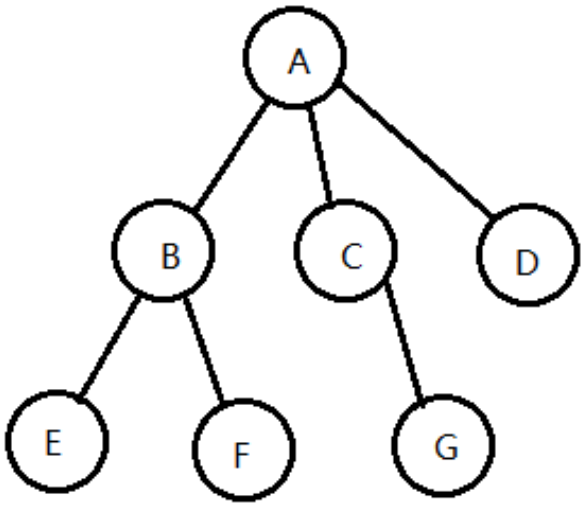
对于插入:数组的插入速度慢,而链表的插入速度快.当我们使用哈希表时,不需要更改数组的结构,只需要在找到对应的数组下标后,进入对应的链表,操作链表即可.所以hash表的整体插入速度也很快.

### 4.1.3. 二叉树(会)

#### 4.1.3.1. 常规概念

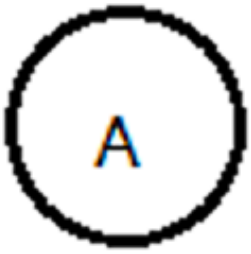
- 树

树是由根结点和若干颗子树构成的。树是由一个集合以及在该集合上定义的一种关系构成的。集合中的元素称为树的结点，所定义的关系称为父子关系。父子关系在树的结点之间建立了一个层次结构。在这种层次结构中有一个结点具有特殊的地位，这个结点称为该树的根结点，或称为树根。注意:普通树中每个节点的子节点个数不一定是2个.



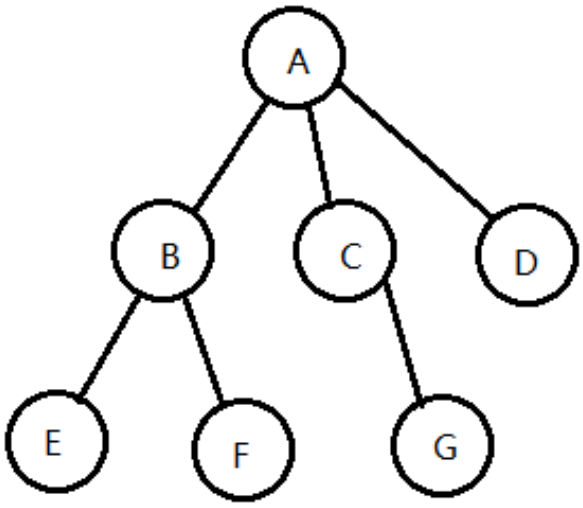

注意:单个结点是一棵树，树根就是该结点本身。

空集合也是树，称为空树。空树中没有结点；




- 森林

森林：由m(m大于等于0)棵互不相交的树的集合称为森林。





- 孩子结点或子结点：一个结点含有的子树称为该结点的子结点；
- 结点的度：一个结点含有的子结点的个数称为该结点的度；
- 叶结点或终端结点：度为0的结点称为叶结点；
- 非终端结点或分支结点：度不为0的结点；
- 双亲结点或父结点：若一个结点含有子结点，则这个结点称为其子结点的父结点；
- 兄弟结点：具有相同父结点的结点互称为兄弟结点;
- 树的度：一棵树中，最大的结点的度称为树的度；
- 结点的层次：从根开始定义起，根为第1层，根的子结点为第2层，以此类推；
- 树的高度或深度：树中结点的最大层次；
- 堂兄弟结点：双亲在同一层的结点互为堂兄弟；
- 结点的祖先：从根到该结点所经分支上的所有结点；
- 子孙：以某结点为根的子树中任一结点都称为该结点的子孙；

#### 4.1.3.2. 树的分类

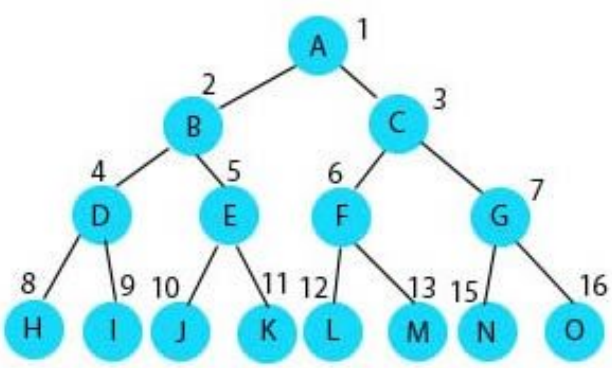
无序树：树中任意节点的子结点之间没有顺序关系，这种树称为无序树,也称为自由树；

有序数：树中任意节点的子结点之间有顺序关系，这种树称为有序树；

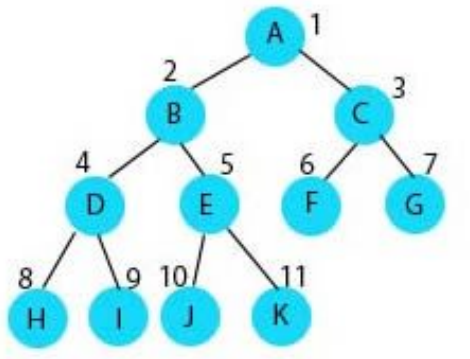
二叉树：每个节点最多含有两个子树的树称为二叉树；

满二叉树：叶节点除外的所有节点均含有两个子树的树被称为满二叉树；

完全二叉树:满二叉树的k-1层的节点数达到最大个数,第k层所有的节点都连续集中在最左边,称为完全二叉树；




(a)满二叉树



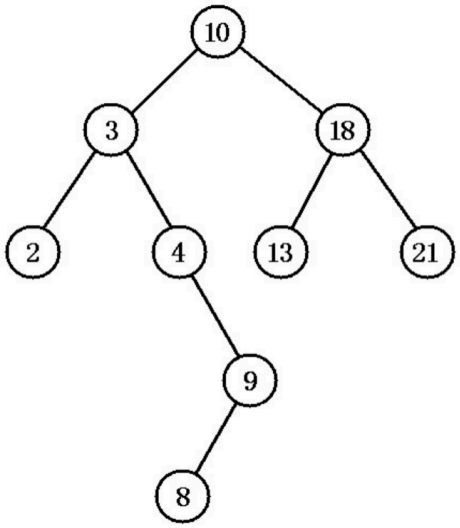

(b)完全二叉树

哈夫曼树（最优二叉树）：带权路径最短的二叉树称为哈夫曼树或最优二叉树。

#### 4.1.3.3. 二叉树遍历

- 二叉树是一种非常重要的数据结构，它同时具有数组和链表各自的特点：它可以像数组一样快速查找，也可以像链表一样快速添加。但是他也有自己的缺点：删除操作复杂。
- 二叉树：是每个结点最多有两个子树的有序树，在使用二叉树的时候，数据并不是随便插入到节点中的，一个节点的左子节点的关键值必须小于此节点，右子节点的关键值必须大于或者是等于此节点，所以又称二叉查找树、二叉排序树、二叉搜索树。
- 二叉树遍历分为三种
    - 先序遍历

        - 首先访问根，再先序遍历左子树，最后先序遍历右子树(根左右)
        - 图中顺序:10-3-2-4-9-8-18-13-21
    - 中序遍历
        - 首先中序遍历左子树，再访问根，最后中序遍历右子树(左根右)
        - 图中顺序:2-3-4-8-9-10-13-18-21
    - 后序遍历
        - 首先后序遍历左子树，再后序遍历右子树，最后访问根(左右根)
        - 图中顺序:2-8-9-4-3-13-21-18-10




**课上练习**

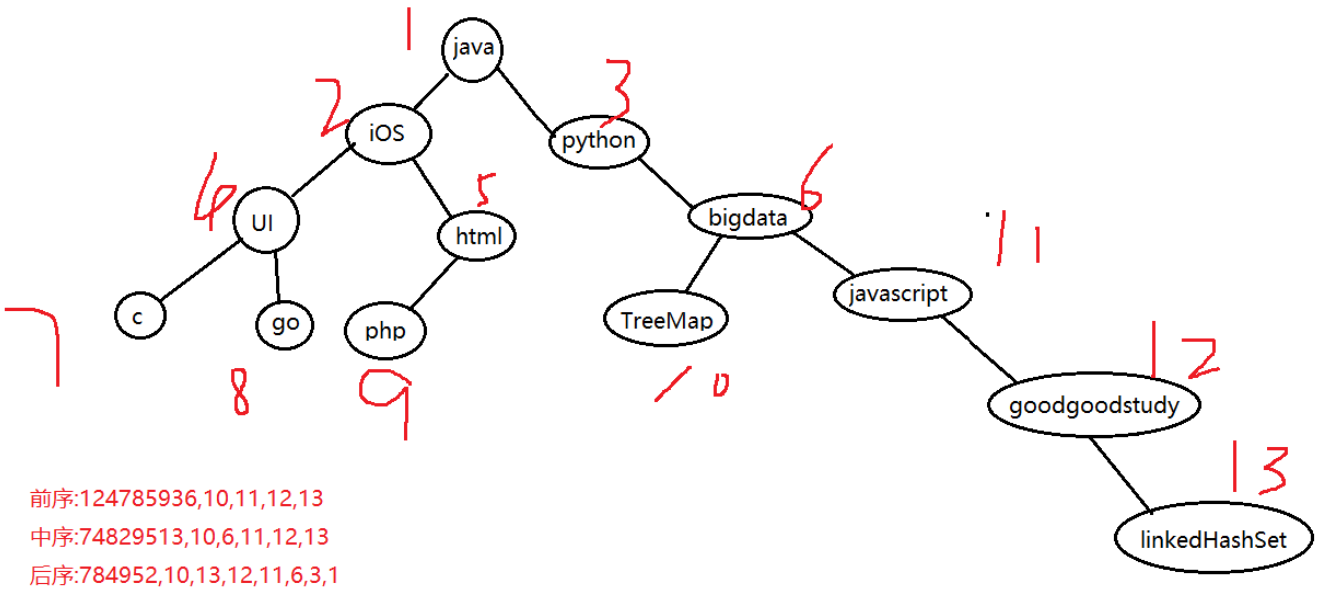
有一组字符串(java,python,iOS,bigdata,html,javascript,php,UI,goodgoodstudy,c,go,linkedHashSet,TreeMap)

要求:

1.按照长度顺序添加到二叉树中,长度相同按照字典排序

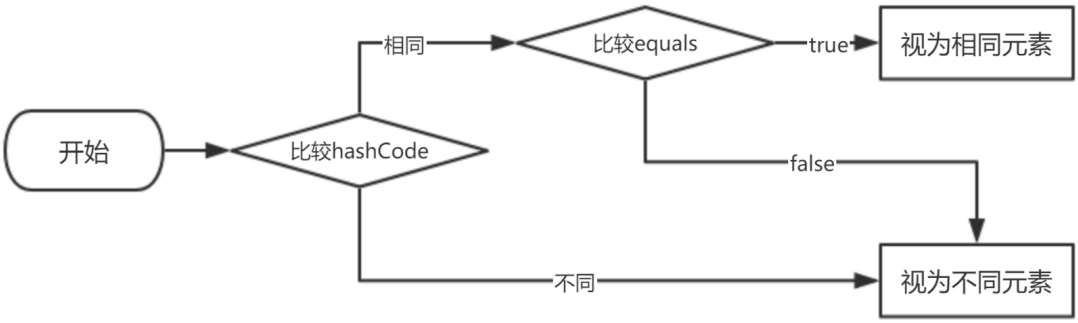
2.分别进行前序,中序,后序遍历

答案参考:




## 4.1.4. HashSet & LinkedHashSet

- 去重原理:




解释:是通过调用元素的hashCode和equals方法实现去重,首先调用hashCode方法,拿到当前对象的哈希码值,去让两个对象的哈希码值进行比较,如果不同,直接认为是两个对象,不再去调用equals,如果相同,再继续调用equals方法,返回true认为是一个对象,返回false认为是两个对象

- 示例代码

```
public class Demo3 {
    public static void main(String[] args) {
        HashSet<String> set = new HashSet<>();
        //说明Set本身的add方法内部实现的去重功能,默认调用的是元素的hashCode和equals方法
```

```
        //String类已经默认重写了hashCode和equals方法
        set.add("java");
        set.add("php");
        set.add("bigdata");
        set.add("html");
        set.add("java");


        System.out.println(set);

        //自己制定的比较规则:并按照年龄和姓名比较,相同则认为是同一个人
        HashSet<Person> set1 = new HashSet<>();
        set1.add(new Person("bing",20));
        set1.add(new Person("bing1",210));
        set1.add(new Person("chenbing",120));
        set1.add(new Person("wangbing",207));
        set1.add(new Person("bing",20));

        System.out.println(set1);
    }
}

class  Person{
    String name;
    int age;

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

    @Override
```

```
37      public String toString() {
38          return "Person{" +
39                  "name='" + name + '\'' +
40                  ", age=" + age +
41                  '}';
42      }
43
44      //自己制定的比较规则:并按照年龄和姓名比较,相同则认为是同一个
    人
45      @Override
46      public boolean equals(Object o) {
47          if (this == o) return true;
48          if (o == null || getClass() != o.getClass())
    return false;
49          Person person = (Person) o;
50          return age == person.age &&
51                  Objects.equals(name, person.name);
52      }
53
54      @Override
55      public int hashCode() {
56          //
57          return name.hashCode()+age*1000;
58      }
59  }
60
```

## 4.1.5. TreeSet(会)

#### 4.1.5.1. TreeSet的简介

TreeSet是一个Set接口的实现类，底层实现是二叉树。这样的集合，会对添加进集合的元素进行去重的处理。 同时， 这个集合会对添加进入的元素进行自动的升序排序。

#### 4.1.5.2. Comparable接口(默认排序)

如果某一个类实现这个接口， 表示自己实现了一个可以和自己的对象进行大小比较的规则。 此时， 这个类的对象就可以直接存储进TreeSet集合中了。 因为此时TreeSet集合已经知道了怎么对两个这个类的对象进行大小比较。

**示例代码**

```
/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description
 */
public class Demo8 {
    public static void main(String[] args) {
      //将字符串存入TreeSet
        /*
         * TreeSet的add方法实现的排序,去重.通过调用元素的compareTo方法
         * String类已经实现了Comparable接口,并重写了compareTo方法
         */
        TreeSet<String> set1 = new TreeSet<>();
        set1.add("java");
        set1.add("php");
```

```
        set1.add("php");
        set1.add("python");
        System.out.println(set1);

      //将Person2对象存入TreeSet
      //要求:人的年龄和姓名比较,年龄和姓名相同是一个人
        TreeSet<Person2> set2 = new TreeSet<>();
        set2.add(new Person2("bing",203));
        set2.add(new Person2("bing1",22));
        set2.add(new Person2("bing2",24));
        set2.add(new Person2("bing",20));

        System.out.println(set2);
    }
}

class  Person2 implements  Comparable<Person2>{
    String name;
    int age;

    public Person2(String name, int age) {
        this.name = name;
        this.age = age;
    }

    @Override
    public String toString() {
        return "Person2{" +
                "name='" + name + '\'' +
                ", age=" + age +
                '}';
    }

```

```
49
50      //自己制定比较规则:人的年龄和姓名比较,年龄和姓名相同是一个人
51      /* @param o 和this进行比较的Person对象
52       * @return 比较结果
53       *       > 0 :   this > o
54       *       ==0 :   this == o
55       *       < 0 :   this < o
56       */
57      @Override
58      public int compareTo(Person2 o) {
59          //先比较年龄
60          int v = this.age-o.age;
61          //再比较姓名
62          return v == 0 ? this.name.compareTo(o.name) :
    v;
63      }
64  }
```

#### 4.1.5.3. Comparator接口(人工排序)

**定义:**使用实现了Comparator接口的compare()方法的比较器对象进行比较

**分析1:**有了Comparable,为什么还要有comparator?

原因:

- 对于自定义的类,代码是我们自己编写的,所有在排序时不管是通过Comparator还是Comparable,排序规则我们都可以自己制定,所以最终使用那种方法没有太大的区别
- 对于系统类,影响非常大.系统类中的代码我们只能用,不能改.这也就意味着系统类内部通过Comparable实现的比较规则已经确定了.这时我们想使用其他的规则对当前的系统类对象进行比较,只能使用Comparator自

己重新制定比较规则.

**分析2:**人工排序和默认排序那个优先级高?

答:人工排序的优先级高于默认排序.

我们可以让TreeSet同时获取到Comparator和Comparable的比较方法,此时对于系统类来说默认排序是系统自带的,通过Comparator实现的人工排序规则是我们想要的,所以系统必须让人工排序优先于默认排序,才能正常的使用后加的排序规则.

**示例代码1**

在下面的案例中

String实现了Comparable接口,使用默认排序.

同时通过实现Comparator接口生成了一个比较器ComWithLength,实现了人工排序,规则是按照字符串的长短比较

```
实现将存储在Treeset中的字符串按照长短比较

分析步骤:
1.生成一个比较器(实现了Comparator接口的类的对象)
2.将比较器作用域TreeSet
```

```
//1.生成一个比较器(实现了Comparator接口的类的对象)
class ComWithLength implements Comparator<String>{
    @Override
    public int compare(String s1, String s2) {
        int num = s1.length()-s2.length();
        return num==0?s1.compareTo(s2):num;
    }
```

```
}
public class Demo5 {
    public static void main(String[] args) {
        //2.将比较器作用域TreeSet
        ComWithLength comWithLength = new ComWithLength();

        TreeSet<String> set = new TreeSet(comWithLength);

        /*
         * TreeSet的add方法实现的排序,去重.通过调用元素的compareTo方法
         * String类已经实现了Comparable接口,并重写了compareTo方法
         */
        set.add("java");
        set.add("php");
        set.add("bigdata");
        set.add("html");
        set.add("java");
    }
}

```

**示例代码2**

Person2实现了Comparable接口,同时通过实现Comparator接口生成了一个比较器ComWithPerson,实现了人工排序,规则都是按照Person2的姓名和年龄比较

```
public class Demo5 {
    public static void main(String[] args) {

```

```
3           //自己制定的比较规则:并按照年龄和姓名比较,相同则认为是同一个人
4           //2.将比较器作用域TreeSet
5       ComWithPerson comWithPerson = new ComWithPerson()
6           TreeSet<Person2> set1 = new TreeSet<>(comWithPerson);
7           set1.add(new Person2("bing",20));
8           set1.add(new Person2("chenbing",210));
9           set1.add(new Person2("chenbing",120));
10          set1.add(new Person2("wangbing",207));
11          set1.add(new Person2("bing",20));
12          set1.add(new Person2("wangbing",207));
13
14          System.out.println(set1);
15      }
16  }
17
18  //1.生成一个比较器(实现了Comparator接口的类的对象)
19  class ComWithPerson implements  Comparator<Person2>{
20      @Override
21      public int compare(Person2 o1, Person2 o2) {
22          int num = o1.name.compareTo(o2.name);
23          return num==0?o1.age-o2.age:num;
24      }
25  }
26
27  class  Person2 implements  Comparable<Person2>{
28
29      String name;
30      int age;
31
32      public Person2(String name, int age) {
33          this.name = name;
```

```
34          this.age = age;
35      }
36
37      @Override
38      public String toString() {
39          return "Person2{" +
40                  "name='" + name + '\'' +
41                  ", age=" + age +
42                  '}';
43      }
44
45      @Override
46      //自己制定的比较规则:并按照年龄和姓名比较,相同则认为是同一个
    人
47      //升序降序?
48      //当前面的对象属性值-后面对象的属性值    结果升序 ,反之降序
49      public int compareTo(Person2 person1) {
50          //先比姓名在比较年龄
51          //int num = name.compareTo(person1.name);
52          int num = person1.name.compareTo(name);
53          return num==0?age-person1.age:num;
54      }
55  }
56
57
```

#### 4.1.5.4. Comparable与Comparator的使用场景

- 如果这个对象， 在项目中大多数的情况下， 都采用相同的大小比较的方式。 比如： 一个Person类， 在大多数情况下， 都是按照年龄进行大小比较的。 此时就可以让Person类实现Comparable接口。

- 如果某一个类的对象， 在临时进行大小比较的时候， 使用的与默认的比较不一样的规则。 比如： 一个Person类， 大多数情况下， 都是使用的年龄进行大小比较的， 但是临时需要使用身高进行一次比较， 此时就可以使用 Comparator 临时完成了。 而且， Comparator的优先级要高于Comparable。
- 系统类想实现新的比较规则,使用Comparator

#### 4.1.5.5. TreeSet的去重

无论使用Comparator还是Comparable，如果两个对象进行大小比较的结果是0， 此时代表这两个对象是相同的对象。 在TreeSet中会完成排重的处理。

**注意：** TreeSet中元素的去重只与对象的大小比较结果有关。 与hashCode()、equals()， 没有任何关系。

# 4.2. Map集合(会)

## 4.2.1. Map API

| 返回值 | 方法 | 描述 |
| --- | --- | --- |
| V | put(K key, V value) | 将一个键值对插入到集合中。 在Map集合中，不允许出现重复的键。 如果添加的键重复了，会用新的值覆盖掉原来的值。并返回被覆盖的原来的值。 |
| V | putIfAbsent（K key, V | 将一个键值对插入到集合中。 向集合中添加元素的时候，如果这个键已经存在了，则不进行添 |

| | | |
|---|---|---|
| | value) | 加。 返回集合中已经存在的这个键对应的值。 |
| void | putAll（Map<K, V> map) | 将一个Map集合中所有的键值对添加到当前集合中。 |
| V | remove(Object key) | 通过键，删除一个键值对，并返回这个被删除的键值对中的值。如果这个键不存在，则返回null。 |
| boolean | remove(Object key, Object value) | 通过键值对进行删除。 只有当键和值一一匹配的时候， 才会进行删除。 |
| void | clear() | 清空集合。 |
| V | replace(K key, V value) | 修改指定的键对应的值， 并返回被覆盖的值。 |
| boolean | replace(K key, V oldValue, V newValue) | 只有当key和oldValue是匹配的情况下，才会将值修改成newValue。 |
| void | replaceAll(BiFunction<K, V, V> biFunction) | 对集合中的元素进行批量的替换将集合中的每一个键值对，带入到BiFunction的方法中, 使用接口方法的返回值替换集合中原来的值。 |
| V | get(K key) | 通过键， 获取值。 如果键不存在， 返回null。 |
| V | getOrDefault（K key) | 通过键， 获取值。 如果键不存在， 返回默认的值。 |
| | | |

| | | |
|---|---|---|
| int | size() | 获取集合中的元素数量（有多少个键值对） |
| boolean | isEmpty() | 判断集合是否为空 |
| boolean | containsKey(K key) | 判断是否包含指定的键 |
| boolean | containsValue(V value) | 判断是否包含指定的值 |
| Set | keySet() | 获取由所有的键组成的集合（因为键是不允许重复的， 因此这里返回的是Set集合） |
| Collection | values() | 获取由所有的值组成的集合 |

### 4.2.2. 示例代码

```
import java.util.*;

/**
 * @Author 千锋大数据教学团队
 * @Company 千锋好程序员大数据
 * @Description Map API
 */
public class MapUsage {
    public static void main(String[] args) {
        // 1. 实例化一个Map集合的实现类对象，并向上转型为接口类型。
        Map<String, String> map = new HashMap<>();

        // 2. 向集合中插入数据
        String value = map.put("name", "xiaoming");
```

```
15          System.out.println(value);       // 由于第一次添加这个键值对，集合中没有被覆盖的值，因此返回null
16          String value2 = map.put("name", "xiaobai");
17          System.out.println(value2);      // 这里是第二次设置name的值，会用xiaobai覆盖掉xiaoming，因此返回xiaoming
18
19          // 3. 向集合中插入数据
20          String value3 = map.putIfAbsent("name", "xiaohong");
21          System.out.println(value3);      // 这里返回的是集合中已经存在的这个键对应的值
22          String value4 = map.putIfAbsent("age", "20");
23          System.out.println(value4);      // 由于这个集合中原来是没有age键存在的，所以返回的是null
24
25          // 4. 将一个Map集合中所有的键值对添加到当前的集合中
26          Map<String, String> tmp = new HashMap<>();
27          tmp.put("height", "177");
28          tmp.put("weight", "65");
29          tmp.put("age", "30");
30          map.putAll(tmp);
31
32
33          // 5. 删除：通过键，删除一个键值对，并返回这个被删除的键值对中的值。
34          String value5 = map.remove("weight");
35          System.out.println(value5);
36
37          // 6. 删除
38          boolean value6 = map.remove("age", "30");
39          System.out.println(value6);
40
41          // 7. 清空集合
```

```
        // map.clear();

        // 8. 修改集合中的某一个键值对（通过键，修改值）
        String value7 = map.replace("name", "xiaohei");
        System.out.println(value7);     // 返回被覆盖的值
        String value8 = map.replace("age", "30");
        System.out.println(value8);     // 由于map中没有age键，因此这个返回null

        // 9. 修改：只有当key和oldValue是匹配的情况下，才会将值修改成newValue。
        boolean value9 = map.replace("name", "xiaohei", "xiaohong");
        System.out.println(value9);

        // 10. 对集合中的元素进行批量的替换
        //     将集合中的每一个键值对，带入到BiFunction的方法中，使用接口方法的返回值替换集合中原来的值。
        map.replaceAll((k, v) -> {
            if (k.equals("height")) {
                return v + "cm";
            }
            return v;
        });

        // 11. 通过键获取值。
        String value10 = map.get("name1");
        System.out.println(value10);
        // 12. 通过键获取值，如果这个键不存在，则返回默认的值。
        String value11 = map.getOrDefault("name1","aaa");
        System.out.println(value11);

```

```
70          // 13. 判断是否包含某一个键
71          boolean value12 = map.containsKey("height");
72          System.out.println(value12);
73
74          boolean value13 = map.containsValue("177");
75          System.out.println(value13);
76
77          // 14. 获取由所有的键组成的Set集合
78          Set<String> keys = map.keySet();
79          //      获取由所有的值组成的Collection集合
80          Collection<String> values = map.values();
81
82          System.out.println(map);
83      }
84  }
```

## 4.2.3. Map集合的遍历(会)

### 4.2.3.1. 使用keySet进行遍历

1. 可以使用keySet()方法获取到集合中所有的键。
2. 遍历存储了所有的键的集合，依次通过键获取值。

```
1  /**
2    * 1. 使用keySet进行遍历
3    * @param map 需要遍历的集合
4    */
5   private static void keyset(Map<String, String> map) {
6       // 1. 获取存储了所有的键的集合
7       Set<String> keys = map.keySet();
8       // 2. 遍历这个Set集合
9       for (String key : keys) {
```

```
            // 2.1. 通过键获取值
            String value = map.get(key);
            // 2.2. 展示一下键和值
            System.out.println("key = " + key + ", value = " + value);
        }
    }
```

#### 4.2.3.2. 使用forEach方法

这个forEach方法， 并不是Iterable接口中的方法。 是Map接口中定义的一个方法。 从功能上将， 与Iterable中的方法差不多。 只是在参数部分有区别。

**default void forEach(BiConsumer<? super K, ? super V> action)**

```
/**
  * 2. 使用forEach进行遍历
  * @param map 需要遍历的集合
  */
private static void forEach(Map<String, String> map) {
    map.forEach((k, v) -> {
        // k: 遍历到的每一个键
        // v: 遍历到的每一个值
        System.out.println("key = " + k + ", value = " + v);
    });
}
```

#### 4.2.3.3. 使用EntrySet进行遍历

**Entry<K, V>:**

是Map中的内部静态接口， 一个Entry对象我们称为一个实体,用来描述集合中的每一个键值对。

```
/**
  * 3. 使用entrySet进行遍历
  * @param map 需要遍历的集合
  */
private static void entrySet(Map<String, String> map) {
    // 1. 获取一个存储有所有的Entry的一个Set集合
    Set<Map.Entry<String, String>> entries = map.entrySet();
    // 2. 遍历Set集合
    for (Map.Entry<String, String> entry : entries) {
        // 2.1. 获取键
        String key = entry.getKey();
        // 2.2. 获取值
        String value = entry.getValue();
        // 2.3. 展示
        System.out.println("key = " + key + ", value = " + value);

         //通过setValue可以去修改原始map的值
         //映射项（键-值对）。Map.entrySet 方法返回映射的 collection 视图，其中的元素属于此类。
         //获得映射项引用的唯一 方法是通过此 collection 视图的迭代器来实现。这些 Map.Entry 对象仅
         //在迭代期间有效；更正式地说，如果在迭代器返回项之后修改了底层映射，则
         //某些映射项的行为是不确定的，除了通过 setValue 在映射项上执行操作之外。
```

```
            //entry.setValue("hello");
        }
    }
```

## 4.2.4. HashMap(了解)

### 4.2.4.1 HashMap基本实现

注意:HashMap可以实现排序:因为他的底层数据结构是由数组+链表+二叉树共同实现的.所以可以排序.同时这样做的目的是提高数据存储的效率.

```
public class Demo8 {
    public static void main(String[] args) {
        HashMap<String, String> map = new HashMap<>();
        map.put("05", "iOS");
        map.put("01", "java");
        map.put("02", "html");
        map.put("03", "BigData");
        map.put("02", "iOS");//会将前面的值覆盖

        System.out.println(map);


        HashMap<Person1, String> map2 = new HashMap<>();
        map2.put(new Person1("bingbing3", 28), "spark");
        map2.put(new Person1("bingbing", 18), "iOS");
        map2.put(new Person1("bingbing2", 118), "html");

        map2.put(new Person1("bingbing", 18), "java");
```

```
19
20          System.out.println(map2);
21      }
22  }
23  class Person1{
24      String name;
25      int age;
26      public Person1() {
27          super();
28          // TODO Auto-generated constructor stub
29      }
30      public Person1(String name, int age) {
31          super();
32          this.name = name;
33          this.age = age;
34      }
35      @Override
36      public String toString() {
37          return "Person1 [name=" + name + ", age=" + age
    + "]";
38      }
39      //重写hashCode方法
40      @Override
41      public int hashCode() {
42          return name.hashCode()+age*1000;
43      }
44      //重写equals方法
45      //@Override
46      public boolean equals(Object obj) {
47          //自己制定比较规则:根据年龄和姓名比较
48          //容错处理
49          if (!(obj instanceof Person1)) {
```

```
50              throw new ClassCastException("当前的对象不是Person1类型的");
51          }
52
53          //向下转型
54          Person1 person = (Person1)obj;
55          return this.name.equals(person.name)  && this.age==person.age;
56      }
57
58  }
```

#### 4.2.4.2 HashMap与Hashtable的区别

1. HashMap是线程不安全的集合， Hashtable是线程安全的集合。
2. HashMap允许出现null键值， Hashtable是不允许的。
3. HashMap的父类是AbstractMap， Hashtable的父类是Dictionary。
4. HashMap的Map接口的新的实现类， 底层算法效率优于Hashtable。

## 4.2.5. TreeMap(了解)

#### 4.2.5.1. 原理实现

与TreeSet一样,进行排列,只是TreeMap是按照键的大小实现,对于值是不管的.我们可以将TreeSet中的值理解成TreeMap中的键.

#### 4.2.5.2. 实现代码

可以自己实现(跟TreeSet类似)

#### 4.2.5.2. 注意点

1. 什么类型的数据类型可以作为key?

- 实现了Comparable接口的compareTo()方法
- 实现了Comparator接口的compare()方法

2. 经常作为key的有:

String,包装类,自定义的实现了要求的类

3. 不可以的代表:数组,ArrayList,LinkedList(如果给他们建立的比较器也可以比较,但是不建议使用)
4. 元素可不可以作为key,跟元素内部的成员有没有关系

```
1
2  public class Demo9 {
3      public static void main(String[] args) {
4          TreeMap<Pig1,String> map = new TreeMap();
5          map.put(new Pig1("荷兰猪"),"java");
6          System.out.println(map);
7      }
8  }
9
10 class Pig1 implements  Comparable<Pig1>{
11     String name;
12     Object object;//没有实现Comparable接口
13     public Pig1(String name) {
14         this.name = name;
15     }
16
17     @Override
18     public int compareTo(Pig1 o) {
19         return 0;
```

```
20      }
21  }
22
```

## 4.2.5. 其他的实现类

- LinkedHashMap
  - 与HashMap类似的，底层多维护了一个链表， 记录每一个键的存储顺序。 也就是说， 在LinkedHashMap中， 键值对的添加顺序可以得到保障。 类似于LinkedHashSet与HashSet。