java集合之Set

一、什么叫哈希码

所谓的哈希码，将是对象按照既定的算法转换成一串数字来表示。

对象和哈希码，就相当于：人和指纹。

用编程的语言来说：就是将一个复杂的对象，转换成一个简单的数据来表示。

hash值就是这个意思。

在java中，hash值包含了对象的值信息，以及地址相关的信息。

二、为什么使用哈希值

场景：
有一个数组，其内保存了100000个元素。假设这些元素都不重复。
现在有一个需求，要求从该数组中取出“Mazi”这个元素。

分析：
　　在数组中，元素和下标有没有直接关系？没有！

因此：
　　传统的用法，遍历数组（使用下标），逐个取出元素进行比较。
　　这种方式，很明显，数组中元素过多的时候，非常慢。

思考：
　　有没有什么方法可以提高这个效率？
　　上述的方法，慢的原因是，元素和下标没有直接关系，而遍历数组，又只能根据下标。

　　有没有一种方式，让数组的元素和下标有直接关系？
　　将对应元素在数组中的下标改为它的哈希值。

这是数据结构中：哈希表的基本原理。

三、Set集合

回顾List：

List是一个基于数组或者链表的集合。特点是：有序、可重复。

Set也是一个基于数组或者链表的集合，特点：无序、不可重复。

所谓的无序，并不是说set集合中的数据是杂乱无章的。而是Set集合，在add元素的时候，会按照元

素的hash值，对元素进行排序。

无序指的是set中元素的顺序，不是放入的时候的顺序，与List的有区别。

List的有序，指的是其内的元素的顺序，就是元素添加的时候的顺序。所以，可以根据你想象中的下标，在List中获取到准确的元素。

```
package com.psfd.util.set;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class SetTest {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("D");
        set.add("B");
        set.add("A");
        set.add("C");

        List<String> list = new ArrayList<>();
        list.add("D");
        list.add("B");
        list.add("A");
        list.add("C");

        //其结果与元素放入的时候的顺序不一致，会根据hashcode进行排序
        System.out.println(set);
        System.out.println(list);


    }
}
```

在java中，所有的Set集合，都衍生于java.util.Set接口。而Set接口又继承了Collection接口。

常用的Set集合有：

HashSet

LinkedHashSet

TreeSet

3.1 HashSet

基于哈希表的集合。

哈希表又是一个基于数组的数据结构。

与一般的数组不同的是：它的下标采用元素的Hash值。

而哈希值又包含了元素在内存中的地址信息，类似于书本的目录。

因此，其查询元素的效率，极高。

HashSet的常用方法：

新增：add、addAll

删除：remove、removeAll

获取大小：size

获取数据：

list中提供了get(index)方法来获取数据，但是在set中，元素本来就是无序的，所以根据下标来获取数据毫无意义。

清空集合：clear

```
package com.psfd.util.set;

import java.util.HashSet;
import java.util.Iterator;
import java.util.Set;

public class SetTest {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("坪山");
        set.add("南山");
        set.add("福田");
        System.out.println(set);

        Set<String> set1 = new HashSet<>();
        set1.add("坑梓");
        set1.add("罗湖");
        set.addAll(set1);
        System.out.println(set);

        Iterator<String> iterator = set.iterator();
        while (iterator.hasNext()) {
            String next = iterator.next();
            System.out.println(next);
        }
        System.out.println("============");
```

```
        for (String string : set) {
            System.out.println(string);
        }


    }
}
```

3.2 LinkedHashSet

LinkedHashSet是HashSet的子类。

LinkedHashSet内部是维护了一个链表来保存数据

采用链表的方式维护了元素的顺序。

```
package com.psfd.util.set;

import java.util.ArrayList;
import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;

public class SetTest {
    public static void main(String[] args) {
        Set<String> set = new HashSet<>();
        set.add("D");
        set.add("B");
        set.add("A");
        set.add("C");

        Set<String> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add("D");
        linkedHashSet.add("B");
        linkedHashSet.add("A");
        linkedHashSet.add("C");

        List<String> list = new ArrayList<>();
        list.add("D");
        list.add("B");
        list.add("A");
        list.add("C");

        System.out.println("HashSet : " + set);
        System.out.println("linkedHashSet : " + linkedHashSet);
        System.out.println("ArrayList : " + list);


    }
```

```
}

执行结果：
HashSet : [A, B, C, D]
linkedHashSet : [D, B, A, C]
ArrayList : [D, B, A, C]
```

3.3 TreeSet

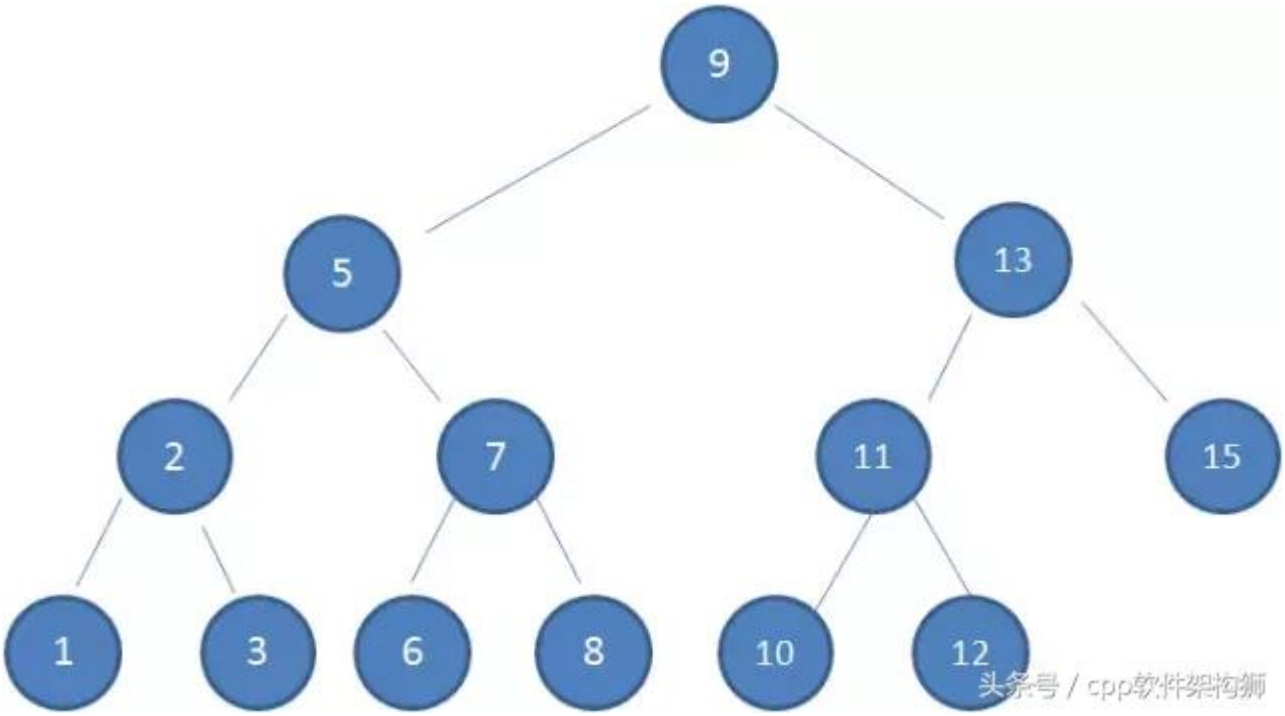
3.3.1 二叉树

二叉树的特点：

1、二叉树是每个节点最多有两个子树的树结构。

2、从根节点开始，左子树上的值，一定小于父节点。

3、右节点的值，一定是大于父节点的值。



当我们需要从二叉树上查找某个节点的时候，逻辑是：

以根节点为例，在上图中，我们要查找12这个元素。

1、找到根节点。根节点的值是9, 12>9

2、忽略左边，查找右边，找到右边的第一个子节点，13

3、12 < 13，因此，找到13这个节点的左边，是11

4、12 > 11，因此，找到11这个元素的右节点

5、发现，该节点就是12，取出12这个元素。

这种查找二叉树的逻辑，就是典型的二分查找法，它的性能是有保障的。

保存数据的逻辑是：

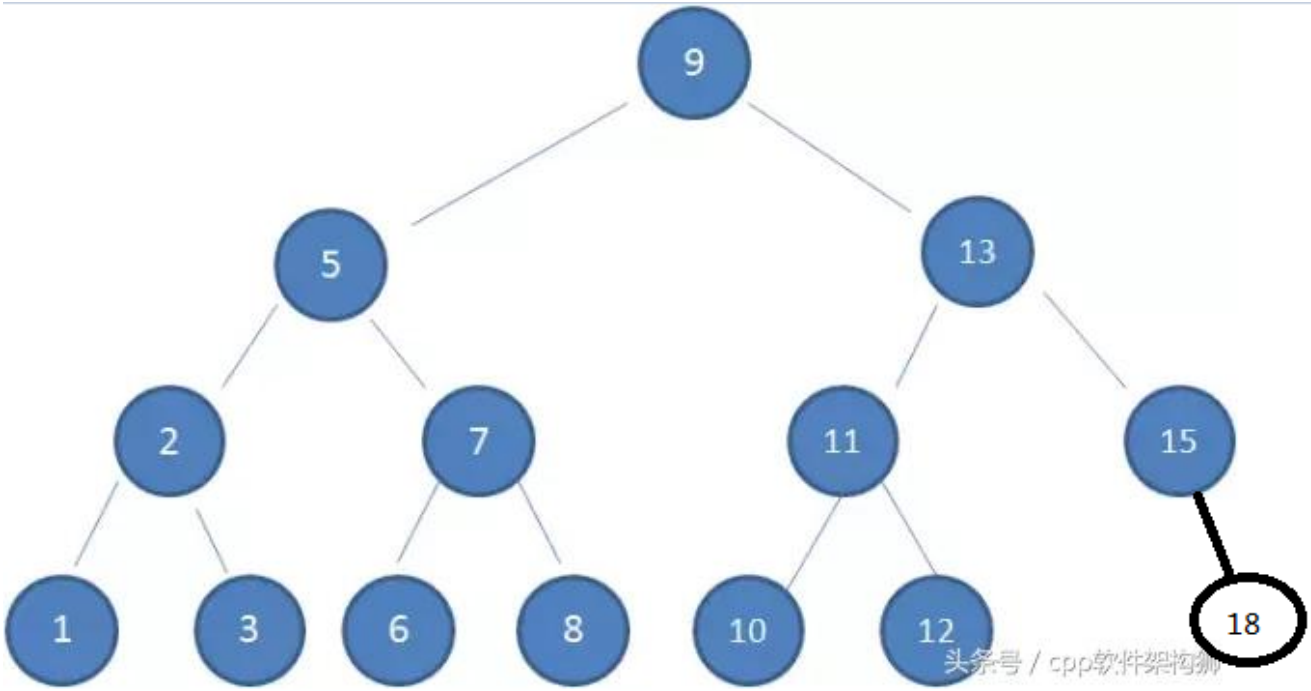
如上图：

要插入数据18.

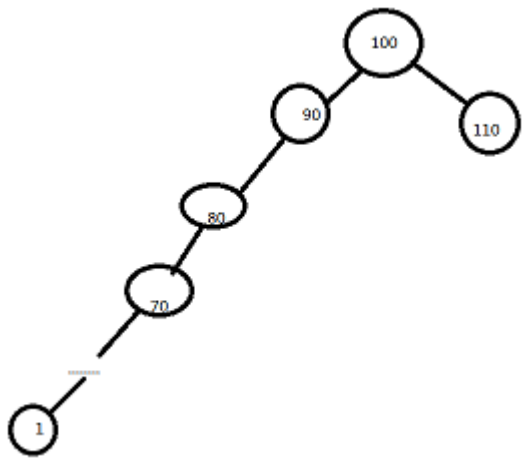
1、与根节点比较，18 > 9，因此，需要方法到右边

2、与右边第一个子节点13比较，18 > 13，因此，需要放到13的右边15

3、18 > 15，因此，需要放到15的右边



分析一种极端情况：



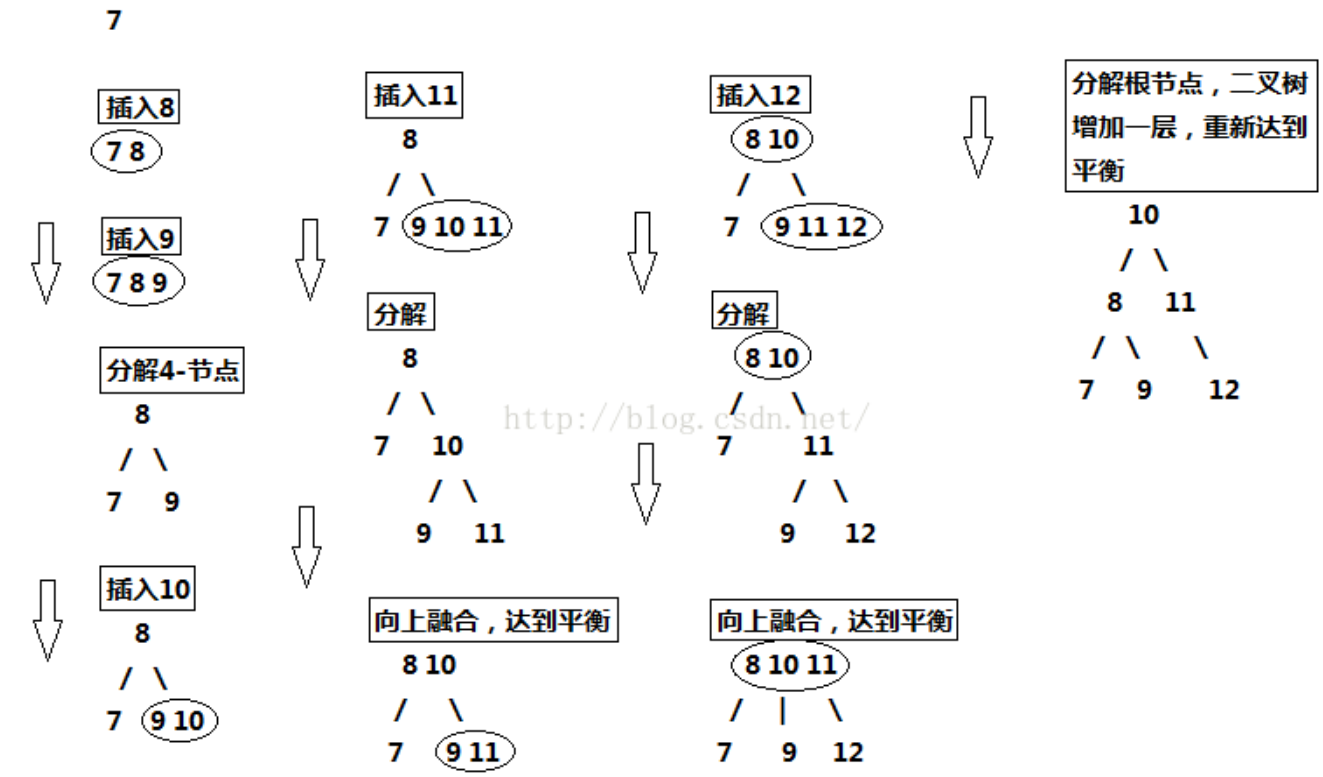
在这种极端情况下，我们要查找二叉树上的某个元素的时候，就变成了线性查找（逐个比较），因此，他的性能会出现问题。

从图上来看，这种情况下，二叉树不平衡，而是成了一个瘸子。有没有一种方式可以避免出现这种“瘸子”的情况？

平衡二叉树。

3.3.2 平衡二叉树

它的左右两个子树的高度差的绝对值不超过1，并且左右两个子树都是一棵平衡二叉树。



通过上述的方式来维护树的平衡。

3.3.3 TreeSet集合

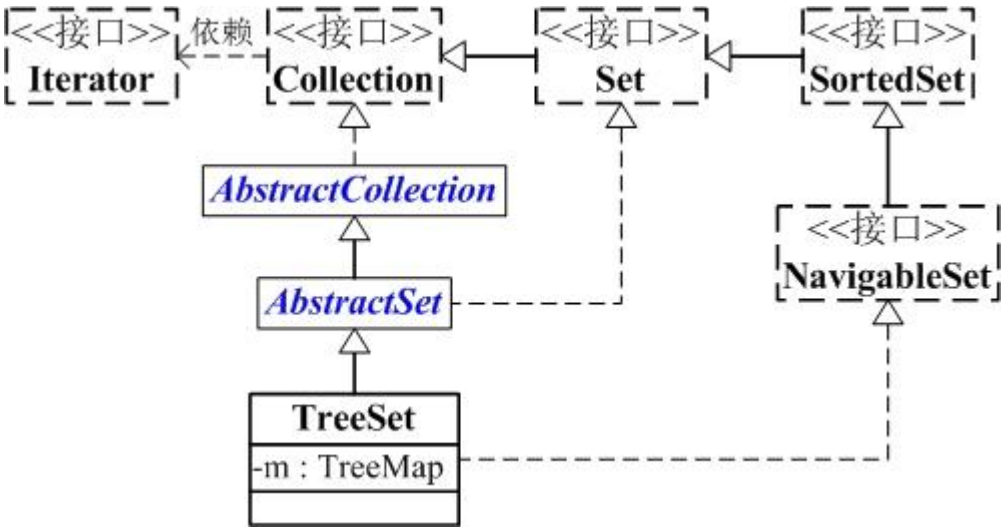
是一种采用红黑树（是一种平衡二叉树）的结构来保存数据的Set集合。采用红黑树的原因是为了保证数据的查询效率。

既然TreeSet是一个Set集合，其内的元素是不重复的。

当你插入一个重复的元素的时候，完全不理。

但是：TreeSet是一种有序的Set集合。

TreeSet的结构



在上图中看到，TreeSet实现了Set接口，还实现了：SortedSet和NavigableSet

SortedSet：是用来排序的

NavigableSet：提供元素导航，更快的实现元素的查找。

TreeSet中除了常规的增删方法之外，还提供了一些特有的方法：

取出大于等于，或者小于等于指定值得元素：

ceiling、highe 获取大于指定元素的最小元素

floorr、lower 获取小于指定元素的最大元素

升序迭代和降序迭代：iterator、descdendingIterator

iterator：升序迭代

descdendingIterator：降序迭代

取出集合中的第一个元素和最后一个元素：first、last

取出并删除第一个或者最后一个元素：

pollFirst、

pollLast

```
TreeSet<String> treeSet = new TreeSet<>();
        treeSet.add("D");
        treeSet.add("B");
        treeSet.add("A");
        treeSet.add("C");
        treeSet.add("F");
        treeSet.add("G");

        Iterator<String> iterator = treeSet.iterator();
```

```
        while (iterator.hasNext()) {
            String next = iterator.next();
            System.out.println(next);
        }

        System.out.println("==================");
        System.out.println(treeSet.pollFirst());

        System.out.println("==================");
        Iterator<String> descendingIterator = treeSet.descendingIterator();
        while (descendingIterator.hasNext()) {
            String next = descendingIterator.next();
            System.out.println(next);
        }
```

3.3.4 TreeSet的自然排序

在创建TreeSet对象的时候，如果采用无参构造方法，那么默认采用的是自然排序。

自然排序放入到TreeSet的对象，一定要实现 java.lang.Comparable接口，并且重写其comparTo方法。

否则就会抛出类型转换异常：

```
Exception in thread "main" java.lang.ClassCastException: com.psfd.util.Student cannot be cast to java.lang.Comparabl
        at java.util.TreeMap.compare(Unknown Source)
        at java.util.TreeMap.put(Unknown Source)
        at java.util.TreeSet.add(Unknown Source)
        at com.psfd.util.set.SetTest.main(SetTest.java:58)
```

```
//实例
package com.psfd.util;

//该javaBean必须要实现Comparable接口，重写其compareTo方法
public class Student implements Comparable<Student> {
    private int age;
    private String name;
    private String sex;

    public int getAge() {
        return age;
    }

    public void setAge(int age) {
        this.age = age;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
```

```
        this.name = name;
    }

    public String getSex() {
        return sex;
    }

    public void setSex(String sex) {
        this.sex = sex;
    }

    @Override
    public int compareTo(Student stu) {
        // 需求：按年龄排序
        return this.age - stu.getAge();
    }

    @Override
    public String toString() {
        return "Student [age=" + age + ", name=" + name + ", sex=" + sex + "]";
    }
}
```

测试代码：

```
package com.psfd.util.set;

import java.util.HashSet;
import java.util.LinkedHashSet;
import java.util.Set;
import java.util.TreeSet;

import com.psfd.util.Student;

public class SetTest {
    public static void main(String[] args) {
        Student stu = new Student();
        stu.setAge(18);
        stu.setName("zhang3");
        stu.setSex("nan");

        Student stu1 = new Student();
        stu1.setAge(16);
        stu1.setName("li4");
        stu1.setSex("nan");

        Student stu2 = new Student();
        stu2.setAge(25);
        stu2.setName("wang5");
        stu2.setSex("nan");
```

```
        Set<Student> hashSet = new HashSet<>();
        hashSet.add(stu);
        hashSet.add(stu1);
        hashSet.add(stu2);

        LinkedHashSet<Student> linkedHashSet = new LinkedHashSet<>();
        linkedHashSet.add(stu);
        linkedHashSet.add(stu1);
        linkedHashSet.add(stu2);

        TreeSet<Student> treeSet = new TreeSet<>();
        treeSet.add(stu);
        treeSet.add(stu1);
        treeSet.add(stu2);

        System.out.println("HashSet : " + hashSet);
        System.out.println("linkedHashSet : " + linkedHashSet);
        System.out.println("TreeSet : " + treeSet);
    }
}
```

结果：

```
HashSet : [Student [age=18, name=zhang3, sex=nan], Student [age=25, name=wang5, sex=nan],
Student [age=16, name=li4, sex=nan]]
linkedHashSet : [Student [age=18, name=zhang3, sex=nan], Student [age=16, name=li4,
sex=nan], Student [age=25, name=wang5, sex=nan]]

TreeSet : [Student [age=16, name=li4, sex=nan], Student [age=18, name=zhang3, sex=nan],
Student [age=25, name=wang5, sex=nan]]
```

从例子中我们得知：

1、放入TreeSet的对象，必须要实现Comparable接口

2、重写compareTo方法

其内的比较逻辑，自己按照需求定制。

返回值的意思：0-相等 ，如果返回零，则视为两个元素相等，就不会将元素加入到集合。

负数：小于

正数：大于

3.3.5TreeSet的定制排序

使用java.util.Comparetor接口。

放入集合的元素不需要实现该接口。采用内部类的方式。

使用的两种方式：

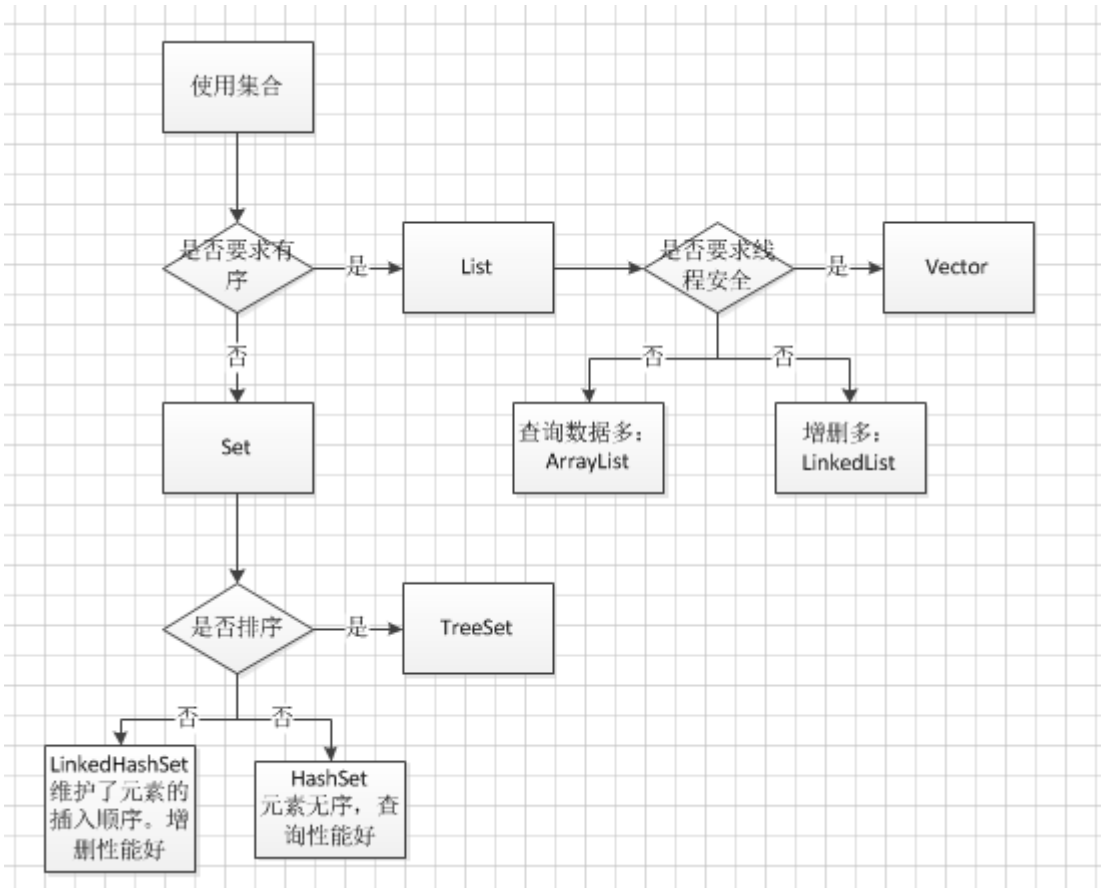
1、写一个比较器的类，实现java.util.Comparetor，重写其compare方法

```
package com.psfd.util.set;

import java.util.Comparator;

import com.psfd.util.Student;

public class StudentComparetor implements Comparator<Student> {

    @Override
    public int compare(Student stu1, Student stu2) {

        return stu1.getAge() - stu2.getAge();
    }

}


测试：
Student stu = new Student();
         stu.setAge(18);
         stu.setName("zhang3");
         stu.setSex("nan");

         Student stu1 = new Student();
         stu1.setAge(16);
         stu1.setName("li4");
         stu1.setSex("nan");

         Student stu2 = new Student();
         stu2.setAge(25);
         stu2.setName("wang5");
         stu2.setSex("nan");

        //声明TreeSet的时候，必须要指定比较器。
         TreeSet<Student> treeSet = new TreeSet<>(new StudentComparetor());
         treeSet.add(stu);
         treeSet.add(stu1);
         treeSet.add(stu2);

         System.out.println("TreeSet : " + treeSet);
```

2、采用匿名内部类的方式

```
TreeSet<Student> treeSet = new TreeSet<>(new Comparator<Student>() {
            @Override
            public int compare(Student stu1, Student stu2) {
                return stu1.getAge() - stu2.getAge();
            }
        });
        treeSet.add(stu);
        treeSet.add(stu1);
        treeSet.add(stu2);
```

四、Set和List的使用场景



总结：

当你知道要使用集合，但是不知道使用哪种具体的集合，那你就用：ArrayList

当你要求元素不重复，那就是使用Set。

在使用Set的时候，如果不知道使用哪种Set，那就使用HashSet

如果要求排序，使用TreeSet

当对元素的唯一性没有要求，就是用List。

查询多，使用ArrayList

增删多，使用LinkedList

线程安全，使用Vector