1.泛型

1.1泛型概述【理解】

- 泛型的介绍

  泛型是JDK5中引入的特性，它提供了编译时类型安全检测机制

- 泛型的好处

  1. 把运行时期的问题提前到了编译期间
  2. 避免了强制类型转换

- 泛型的定义格式
  - <类型>: 指定一种类型的格式.尖括号里面可以任意书写,一般只写一个字母.例如:
  - <类型1,类型2...>: 指定多种类型的格式,多种类型之间用逗号隔开.例如:

1.2泛型类【应用】

- 定义格式

```
修饰符 class 类名<类型> {  }
```

- 示例代码
  - 泛型类

```
public class Generic<T> {
    private T t;

    public T getT() {
        return t;
    }

    public void setT(T t) {
        this.t = t;
    }
}
```

  - 测试类

```
public class GenericDemo1 {
    public static void main(String[] args) {
        Generic<String> g1 = new Generic<String>();
        g1.setT("杨幂");
        System.out.println(g1.getT());

        Generic<Integer> g2 = new Generic<Integer>();
        g2.setT(30);
        System.out.println(g2.getT());
```

```
        Generic<Boolean> g3 = new Generic<Boolean>();
        g3.setT(true);
        System.out.println(g3.getT());
    }
}
```

1.3泛型方法【应用】

- 定义格式

```
修饰符 <类型> 返回值类型 方法名(类型 变量名) {  }
```

- 示例代码
  - 带有泛型方法的类

```
public class Generic {
    public <T> void show(T t) {
        System.out.println(t);
    }
}
```

  - 测试类

```
public class GenericDemo2 {
    public static void main(String[] args) {
         Generic g = new Generic();
        g.show("柳岩");
        g.show(30);
        g.show(true);
        g.show(12.34);
    }
}
```

1.4泛型接口【应用】

- 定义格式

```
修饰符 interface 接口名<类型> {  }
```

- 示例代码
  - 泛型接口

```
public interface Generic<T> {
    void show(T t);
}
```

  - 泛型接口实现类1

    定义实现类时,定义和接口相同泛型,创建实现类对象时明确泛型的具体类型

```
public class GenericImpl1<T> implements Generic<T> {
    @Override
    public void show(T t) {
        System.out.println(t);
    }
}
```

- 泛型接口实现类2

  定义实现类时,直接明确泛型的具体类型

```
public class GenericImpl2 implements Generic<Integer>{
     @Override
     public void show(Integer t) {
          System.out.println(t);
     }
}
```

- 测试类

```
public class GenericDemo3 {
    public static void main(String[] args) {
        GenericImpl1<String> g1 = new GenericImpl<String>();
        g1.show("林青霞");
        GenericImpl1<Integer> g2 = new GenericImpl<Integer>();
        g2.show(30);

        GenericImpl2 g3 = new GenericImpl2();
        g3.show(10);
    }
}
```

1.5类型通配符

- 类型通配符: <?>
  - ArrayList<?>: 表示元素类型未知的ArrayList,它的元素可以匹配任何的类型
  - 但是并不能把元素添加到ArrayList中了,获取出来的也是父类类型
- 类型通配符上限: <? extends 类型>
  - ArrayListList <? extends Number>: 它表示的类型是Number或者其子类型
- 类型通配符下限: <? super 类型>
  - ArrayListList <? super Number>: 它表示的类型是Number或者其父类型
- 泛型通配符的使用

```
public class GenericDemo4 {
    public static void main(String[] args) {
        ArrayList<Integer> list1 = new ArrayList<>();

        ArrayList<String> list2 = new ArrayList<>();
```

```
        ArrayList<Number> list3 = new ArrayList<>();
        ArrayList<Object> list4 = new ArrayList<>();

        method(list1);
        method(list2);
        method(list3);
        method(list4);

        getElement1(list1);
        getElement1(list2);//报错
        getElement1(list3);
        getElement1(list4);//报错

        getElement2(list1);//报错
        getElement2(list2);//报错
        getElement2(list3);
        getElement2(list4);
    }

    // 泛型通配符: 此时的泛型?,可以是任意类型
    public static void method(ArrayList<?> list){}
    // 泛型的上限: 此时的泛型?,必须是Number类型或者Number类型的子类
    public static void getElement1(ArrayList<? extends Number> list){}
    // 泛型的下限: 此时的泛型?,必须是Number类型或者Number类型的父类
    public static void getElement2(ArrayList<? super Number> list){}

}
```

2.Set集合

2.1Set集合概述和特点【应用】

- 不可以存储重复元素
- 没有索引,不能使用普通for循环遍历

2.2Set集合的使用【应用】

存储字符串并遍历

```
public class MySet1 {
    public static void main(String[] args) {
        //创建集合对象
        Set<String> set = new TreeSet<>();
        //添加元素
        set.add("ccc");
        set.add("aaa");
        set.add("aaa");
        set.add("bbb");

//        for (int i = 0; i < set.size(); i++) {
//            //Set集合是没有索引的，所以不能使用通过索引获取元素的方法
//        }
```

```
        //遍历集合
        Iterator<String> it = set.iterator();
        while (it.hasNext()){
            String s = it.next();
            System.out.println(s);
        }
        System.out.println("----------------------------------");
        for (String s : set) {
            System.out.println(s);
        }
    }
}
```

3.TreeSet集合

3.1TreeSet集合概述和特点【应用】

- 不可以存储重复元素
- 没有索引
- 可以将元素按照规则进行排序
    - TreeSet()：根据其元素的自然排序进行排序
    - TreeSet(Comparator comparator)： 根据指定的比较器进行排序

3.2TreeSet集合基本使用【应用】

存储Integer类型的整数并遍历

```
public class TreeSetDemo01 {
    public static void main(String[] args) {
        //创建集合对象
        TreeSet<Integer> ts = new TreeSet<Integer>();

        //添加元素
        ts.add(10);
        ts.add(40);
        ts.add(30);
        ts.add(50);
        ts.add(20);

        ts.add(30);

        //遍历集合
        for(Integer i : ts) {
            System.out.println(i);
        }
    }
}
```

3.3自然排序Comparable的使用【应用】

- 案例需求
  - 存储学生对象并遍历，创建TreeSet集合使用无参构造方法
  - 要求：按照年龄从小到大排序，年龄相同时，按照姓名的字母顺序排序
- 实现步骤
  1. 使用空参构造创建TreeSet集合
     - 用TreeSet集合存储自定义对象，无参构造方法使用的是自然排序对元素进行排序的
  2. 自定义的Student类实现Comparable接口
     - 自然排序，就是让元素所属的类实现Comparable接口，重写compareTo(T o)方法
  3. 重写接口中的compareTo方法
     - 重写方法时，一定要注意排序规则必须按照要求的主要条件和次要条件来写
- 代码实现

  学生类

```
public class Student implements Comparable<Student>{
    private String name;
    private int age;

    public Student() {
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

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

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

    }
```

```
    @Override
    public int compareTo(Student o) {
        //按照对象的年龄进行排序
        //主要判断条件：按照年龄从小到大排序
        int result = this.age - o.age;
        //次要判断条件：年龄相同时，按照姓名的字母顺序排序
        result = result == 0 ? this.name.compareTo(o.getName()) : result;
        return result;
    }
}
```

测试类

```
public class MyTreeSet2 {
    public static void main(String[] args) {
        //创建集合对象
        TreeSet<Student> ts = new TreeSet<>();
        //创建学生对象
        Student s1 = new Student("zhangsan",28);
        Student s2 = new Student("lisi",27);
        Student s3 = new Student("wangwu",29);
        Student s4 = new Student("zhaoliu",28);
        Student s5 = new Student("qianqi",30);
        //把学生添加到集合
        ts.add(s1);
        ts.add(s2);
        ts.add(s3);
        ts.add(s4);
        ts.add(s5);
        //遍历集合
        for (Student student : ts) {
            System.out.println(student);
        }
    }
}
```

3.4比较器排序Comparator的使用【应用】

- 案例需求
  - 存储老师对象并遍历，创建TreeSet集合使用带参构造方法
  - 要求：按照年龄从小到大排序，年龄相同时，按照姓名的字母顺序排序
- 实现步骤
  - 用TreeSet集合存储自定义对象，带参构造方法使用的是比较器排序对元素进行排序的
  - 比较器排序，就是让集合构造方法接收Comparator的实现类对象，重写compare(T o1,T o2)方法
  - 重写方法时，一定要注意排序规则必须按照要求的主要条件和次要条件来写
- 代码实现

  老师类

```
public class Teacher {
    private String name;
    private int age;

    public Teacher() {
    }

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

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public int getAge() {
        return age;
    }

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

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

测试类

```
public class MyTreeSet4 {
    public static void main(String[] args) {
        //创建集合对象
        TreeSet<Teacher> ts = new TreeSet<>(new Comparator<Teacher>() {
            @Override
            public int compare(Teacher o1, Teacher o2) {
                //o1表示现在要存入的那个元素
                //o2表示已经存入到集合中的元素

                //主要条件
                int result = o1.getAge() - o2.getAge();
                //次要条件

                result = result == 0 ? o1.getName().compareTo(o2.getName()) : result;
```

```
                return result;
            }
        });
         //创建老师对象
        Teacher t1 = new Teacher("zhangsan",23);
        Teacher t2 = new Teacher("lisi",22);
        Teacher t3 = new Teacher("wangwu",24);
        Teacher t4 = new Teacher("zhaoliu",24);
         //把老师添加到集合
        ts.add(t1);
        ts.add(t2);
        ts.add(t3);
        ts.add(t4);
         //遍历集合
        for (Teacher teacher : ts) {
            System.out.println(teacher);
        }
    }
}
```

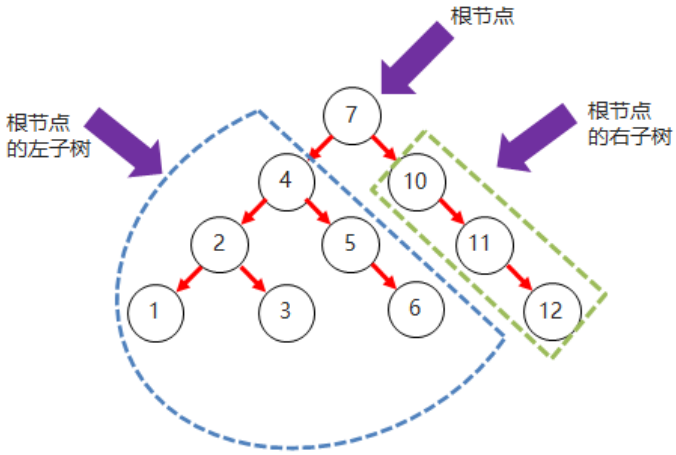
3.5两种比较方式总结【理解】

- 两种比较方式小结
  - 自然排序: 自定义类实现Comparable接口,重写compareTo方法,根据返回值进行排序
  - 比较器排序: 创建TreeSet对象的时候传递Comparator的实现类对象,重写compare方法,根据返回值进行排序
  - 在使用的时候,默认使用自然排序,当自然排序不满足现在的需求时,必须使用比较器排序
- 两种方式中关于返回值的规则
  - 如果返回值为负数，表示当前存入的元素是较小值，存左边
  - 如果返回值为0，表示当前存入的元素跟集合中元素重复了，不存
  - 如果返回值为正数，表示当前存入的元素是较大值，存右边

4.数据结构

4.1二叉树【理解】

- 二叉树的特点
  - 二叉树中,任意一个节点的度要小于等于2
    - 节点: 在树结构中,每一个元素称之为节点
    - 度: 每一个节点的子节点数量称之为度
- 二叉树结构图

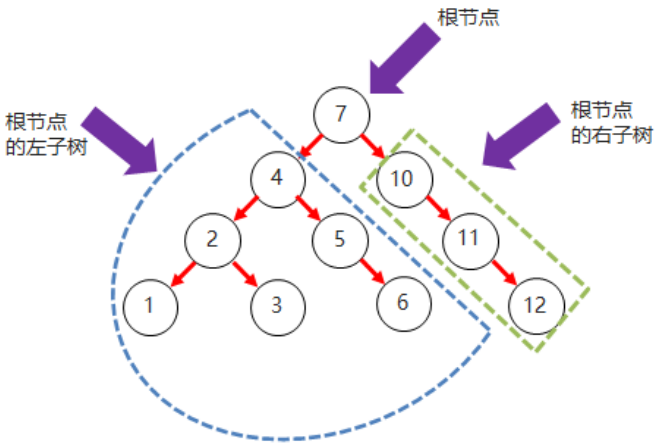


最顶层的为：根节点
4节点为7节点的左子节点
10节点为7节点的右子节点

蓝色虚线：根节点的左子树
根节点的左子树高：3
绿色虚线：根节点的右子树
根节点的右子树高：3

4.2二叉查找树【理解】

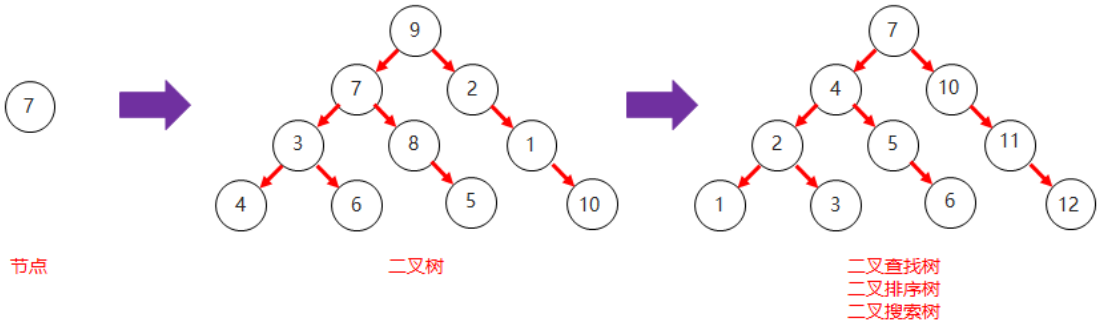
- 二叉查找树的特点
  - 二叉查找树,又称二叉排序树或者二叉搜索树
  - 每一个节点上最多有两个子节点
  - 左子树上所有节点的值都小于根节点的值
  - 右子树上所有节点的值都大于根节点的值
- 二叉查找树结构图



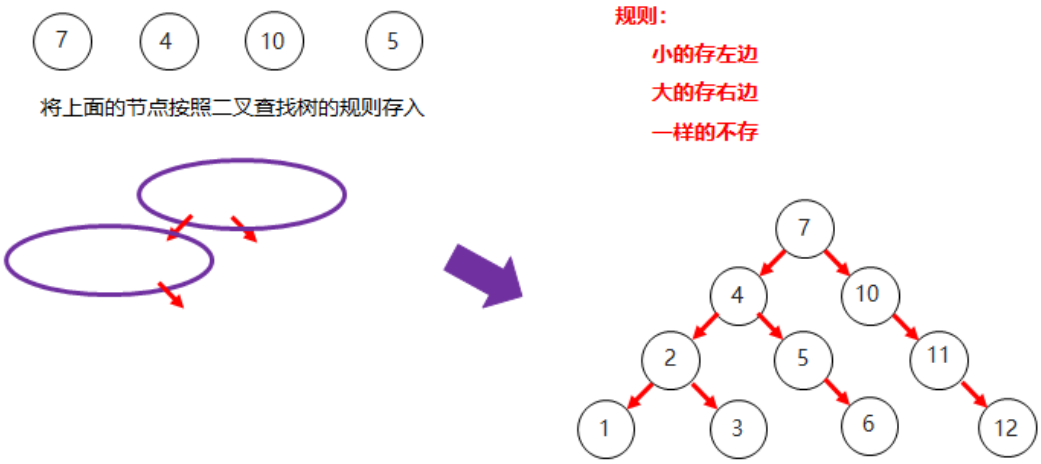
整个树高：4
最顶层的为：根节点
蓝色虚线：根节点的左子树（左孩子）
根节点的左子树高：3
绿色虚线：根节点的右子树（右孩子）
根节点的右子树高：3

这颗树中，根节点左边的都比根节点小
根节点右边的都比根节点大，所以是二叉查找树。

- 二叉查找树和二叉树对比结构图

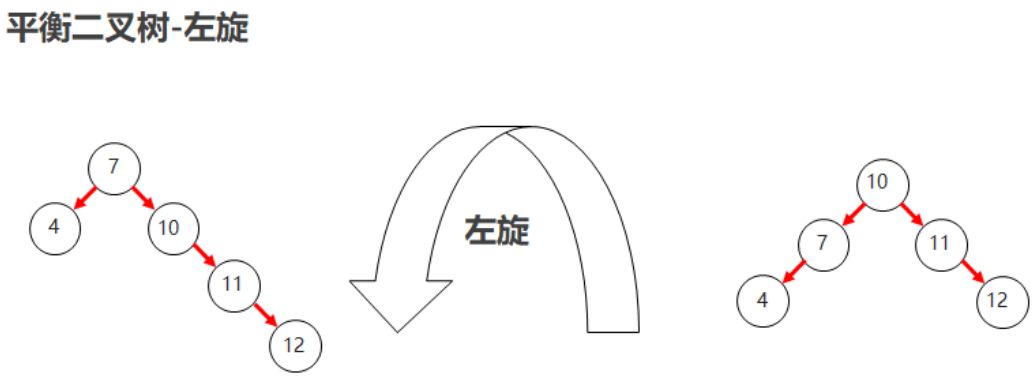


- 二叉查找树添加节点规则
  - 小的存左边
  - 大的存右边
  - 一样的不存

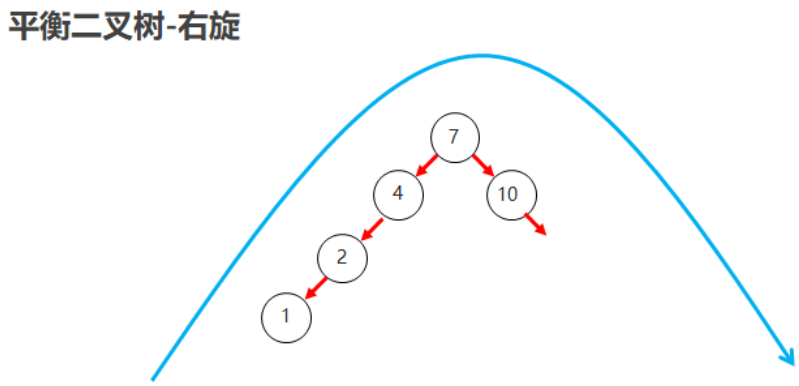


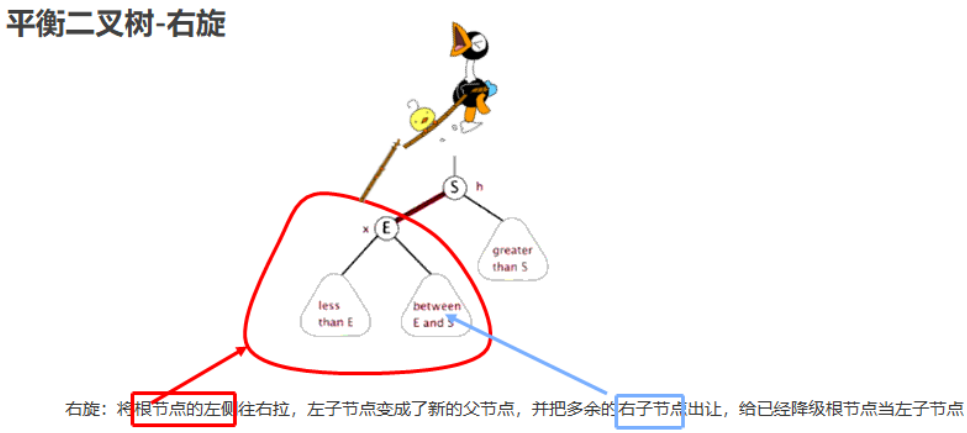
4.3平衡二叉树【理解】

- 平衡二叉树的特点
  - 二叉树左右两个子树的高度差不超过1
  - 任意节点的左右两个子树都是一颗平衡二叉树
- 平衡二叉树旋转
  - 旋转触发时机
    - 当添加一个节点之后,该树不再是一颗平衡二叉树
  - 左旋
    - 就是将根节点的右侧往左拉,原先的右子节点变成新的父节点,并把多余的左子节点出让,给已经降级的根节点当右子节点

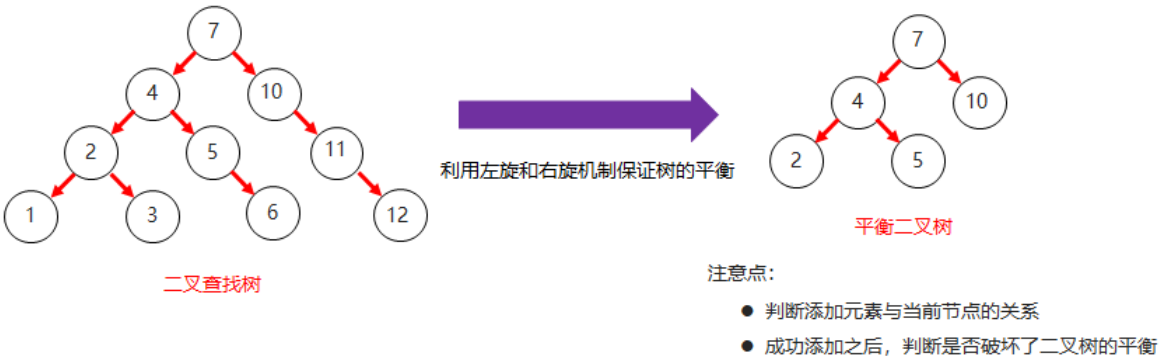


- 右旋
  - 就是将根节点的左侧往右拉,左子节点变成了新的父节点,并把多余的右子节点出让,给已经降级根节点当左子节点

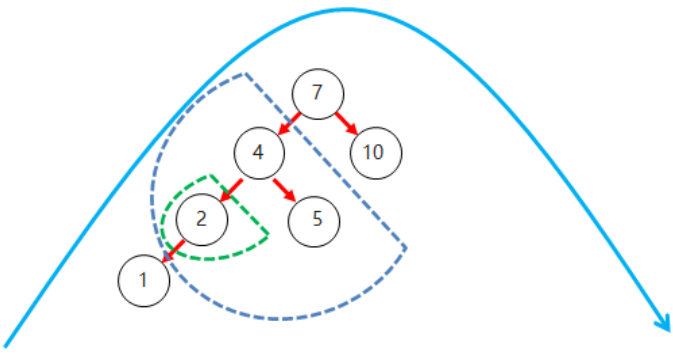




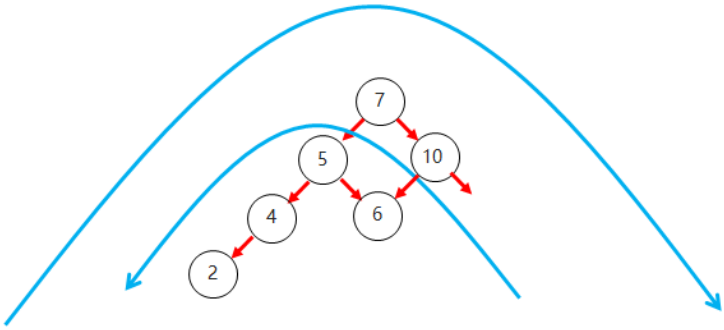
- 平衡二叉树和二叉查找树对比结构图



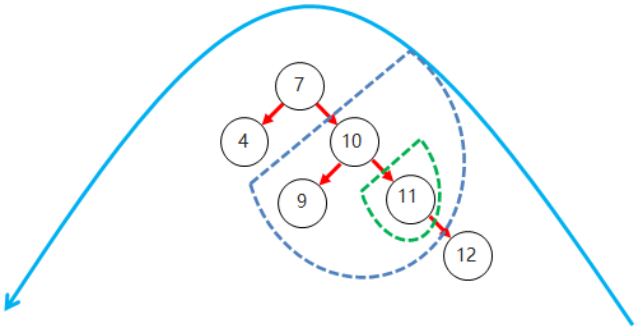
- 平衡二叉树旋转的四种情况
  - 左左
    - 左左: 当根节点左子树的左子树有节点插入,导致二叉树不平衡
    - 如何旋转: 直接对整体进行右旋即可



  - 左右
    - 左右: 当根节点左子树的右子树有节点插入,导致二叉树不平衡
    - 如何旋转: 先在左子树对应的节点位置进行左旋,在对整体进行右旋



- 右右
  - 右右: 当根节点右子树的右子树有节点插入,导致二叉树不平衡
  - 如何旋转: 直接对整体进行左旋即可



- 右左
  - 右左:当根节点右子树的左子树有节点插入,导致二叉树不平衡
  - 如何旋转: 先在右子树对应的节点位置进行右旋,在对整体进行左旋