1.能排序的二叉树：二叉排序树（BST）

二叉排序树（Binary Sorted Tree）是二叉树的另一个典型应用

之所以称之为二叉排序树，是因为这种二叉树结构在创建的时候遵循一些特殊的规则，并且在对这棵二叉树进行指定顺序的遍历之后

能够得到一个有序序列。至于创建二叉排序树的规则和遍历二叉排序树的方式，我们会在后面介绍。

也就是说，二叉排序树相当于是另一种排序算法的实现。但是，二叉排序树也同样存在着不足：二叉排序树中不允许存在取值相同的值

这也是由其结构决定的

下面，我们就以Java中的TreeSet结构为入手点，一起来分析一下二叉排序树的结构和实现原理

①回忆杀：TreeSet

TreeSet是Java集合框架下，Set接口的一种实现类。它的接口实现关系是：

Interface Collection -> Interface Set -> Interface SortedSet -> Class TreeSet

TreeSet作为一种二叉排序树，在使用的时候，具有如下要求：

1.添加到TreeSet中的元素必须可以进行自然排序

2.或者为TreeSet加载一个能够为其中元素进行定制排序的排序器

上面两个要求，都离不开“比较”二字。也就是说，TreeSet要求能够对集合中的元素进行“大小”比较，并且能够得到int层面的大小比较结果

只有TreeSet集合中的元素之间能够互相比较大小，TreeSet才能够对其中的元素进行排序操作

而TreeSet能够接受的元素比较方式有两种：元素与元素之间自发进行大小比较：自然排序，和通过排序器对元素与元素进行大小比较：定制排序

下面让我们一起回忆一下这两种实现方式

1.自然排序

自然排序要求加入TreeSet中的元素对应数据类型实现Comparable<T>接口，并在接口的抽象方法int compareTo(T t)中实现当前元素和参数t元素之间的大小关系比较

比较结果必须是int类型的，而TreeSet关心的并不是这个int类型返回值的取值，而是其正负性：

返回值 > 0：表示this > t

返回值 == 0：表示this == t

返回值 < 0：表示this < t

例如：现在要向一个TreeSet中添加自定义类型Person的对象，Person类定义如下：

```
public class Person {

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

    //构造器、get/set方法省略不写

}
```

并且要求TreeSet中的Person对象之间按照年龄age属性进行自然排序，则需要改变Person类为如下结构：

```
public class Person implements Comparable<Person> {

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

    @Override
    public int compareTo(Person o) {
        return this.age - o.age;
    }

```

```
}
```

其中的compareTo()方法是接口Comparable的抽象方法，在TreeSet内部，TreeSet正是调用这个方法来比较各个Person对象之间的“大小”的

自然排序的实现简单明了，但是也有不足：同一个类型的对象之间只能够通过自然排序定义一种比较大小的方式，不够灵活

于是乎TreeSet也能够接受另一种比较方式：定制排序

2.定制排序

定制排序的实现，需要我们定义一个第三方的类——排序器类

不同的排序器类型，都要实现Comparator<T>接口，并且实现其中int compare(T o1, T o2)方法

这个方法的作用与Comparable接口中的compareTo方法相似，都是用来比较两个对象o1和o2之间的大小

并且以int值正负的方式进行返回。当向TreeSet中添加元素的时候，TreeSet会在内部调用指定的比较器对象，

通过这个方法比较新加入的对象和原有的对象之间的大小关系，进一步确定新加入元素是否能够在TreeSet中存在（别忘了TreeSet 不允许存储大小相同的两个值）

或者新加入元素应该存储在什么位置上

假如现有一个学生类Student，定义结构如下：

```
public class Student {

    private String name;
    private String id;

    private int math;
    private int chinese;
    private int english;

}
```

那么我们可以针对这个学生类定义很多不同的比较器，分别实现对学生按照数学成绩、语文成绩、英语成绩进行TreeSet排序的操作：

数学成绩比较器：

```

```

```java
/**
 * 学生类数学成绩比较器
 * 能够是的TreeSet对Student对象按照数学成绩属性math进行排序
 */
public class MathComparator implements Comparator<Student> {

    @Override
    public int compare(Student o1, Student o2) {
        return o1.getMath() - o2.getMath();
    }

}
```

语文成绩比较器：

```java
/**
 * 学生类语文成绩比较器
 * 能够是的TreeSet对Student对象按照语文成绩属性chinese进行排序
 */
public class ChineseComparator implements Comparator<Student> {

    @Override
    public int compare(Student o1, Student o2) {
        return o1.getChinese() - o2.getChinese();
    }

}
```

英语成绩比较器：

```java
/**
 * 学生类英语成绩比较器
 * 能够是的TreeSet对Student对象按照英语成绩属性english进行排序
 */
public class EnglishComparator implements Comparator<Student> {

    @Override
    public int compare(Student o1, Student o2) {
        return o1.getEnglish() - o2.getEnglish();
    }
```

```
11
12 }
```

比较器接口实现的定制排序功能的优点显而易见：通过实现不同的比较器，就能够对同一数据类型的对象之间按照不同的标准进行排序和比较

但是每一次都要创建一个比较器的实现类未免太过麻烦，所以到底是使用自然排序还是定制排序，还是要具体问题具体分析的

3.自然排序和定制排序的总结：

用一个比喻来说明自然排序和定制排序：

大学生军训，如果让学生自发的进行排队，那么学生与学生之间可能会默认的按照身高进行比较，并决定站位，这属于自然排序：元素与元素之间自发进行比较

如果让教官对学生进行排队操作，有的教官可能会让学生按照身高排队；有的教官可能让学生按照班级编号排队；有的教官可能让学生按照学号排队……

总之不同的教官有不同的排序标准，这就属于定制排序，学生就是加入TreeSet中的元素，而不同的教官就是不同的排序器

下面是一组测试TreeSet用法的代码：

```java
public class TestTreeSet {

    public static void main(String[] args) {

        //自然排序
        Person p1 = new Person("张三", 22, "111111");
        Person p2 = new Person("李四", 21, "222222");
        Person p3 = new Person("王五", 23, "333333");
        Person p4 = new Person("陈六", 20, "444444");
        Person p5 = new Person("田七", 22, "555555");

        TreeSet<Person> ts1 = new TreeSet<>();
        ts1.add(p1);
        ts1.add(p2);
        ts1.add(p3);
        ts1.add(p4);
        ts1.add(p5);

        //在ts1中不存在田七这个人，因为田七的年龄和张三的年龄相同，且张三先加入
        System.out.println(ts1);

```

```
        //-------------------------------------------------------------------

        //定制排序
        Student s1 = new Student("张三", "111111", 98, 95, 100);
        Student s2 = new Student("李四", "222222", 66, 57, 97);
        Student s3 = new Student("王五", "333333", 87, 86, 55);
        Student s4 = new Student("陈六", "444444", 52, 73, 81);
        Student s5 = new Student("田七", "555555", 99, 28, 90);

        MathComparator mc = new MathComparator();
        TreeSet<Student> ts2 = new TreeSet<>(mc);  //按照数学排序
        ts2.add(s1);
        ts2.add(s2);
        ts2.add(s3);
        ts2.add(s4);
        ts2.add(s5);
        System.out.println(ts2);

        ChineseComparator cc = new ChineseComparator();
        TreeSet<Student> ts3 = new TreeSet<>(cc);  //按照语文成绩排序
        ts3.add(s1);
        ts3.add(s2);
        ts3.add(s3);
        ts3.add(s4);
        ts3.add(s5);
        System.out.println(ts3);

        EnglishComparator ec = new EnglishComparator();
        TreeSet<Student> ts4 = new TreeSet<>(ec);
        ts4.add(s1);
        ts4.add(s2);
        ts4.add(s3);
        ts4.add(s4);
        ts4.add(s5);
        System.out.println(ts4);

    }

}
```

通过上述案例我们发现：TreeSet的默认排序方式都是升序排序的，如果我们想要元素进行降序排序，只要将compareTo()方法或者compare()方法中

减号两边的元素进行互换，就能够实现降序排序的操作

②二叉排序树的构建规范

TreeSet内部的实现原理实际上是使用了二叉排序树的一种升级版本——红黑树结构

红黑树我们会在后面的章节中进行讲解，现在我们先来研究一下节本的二叉排序树的构建方式

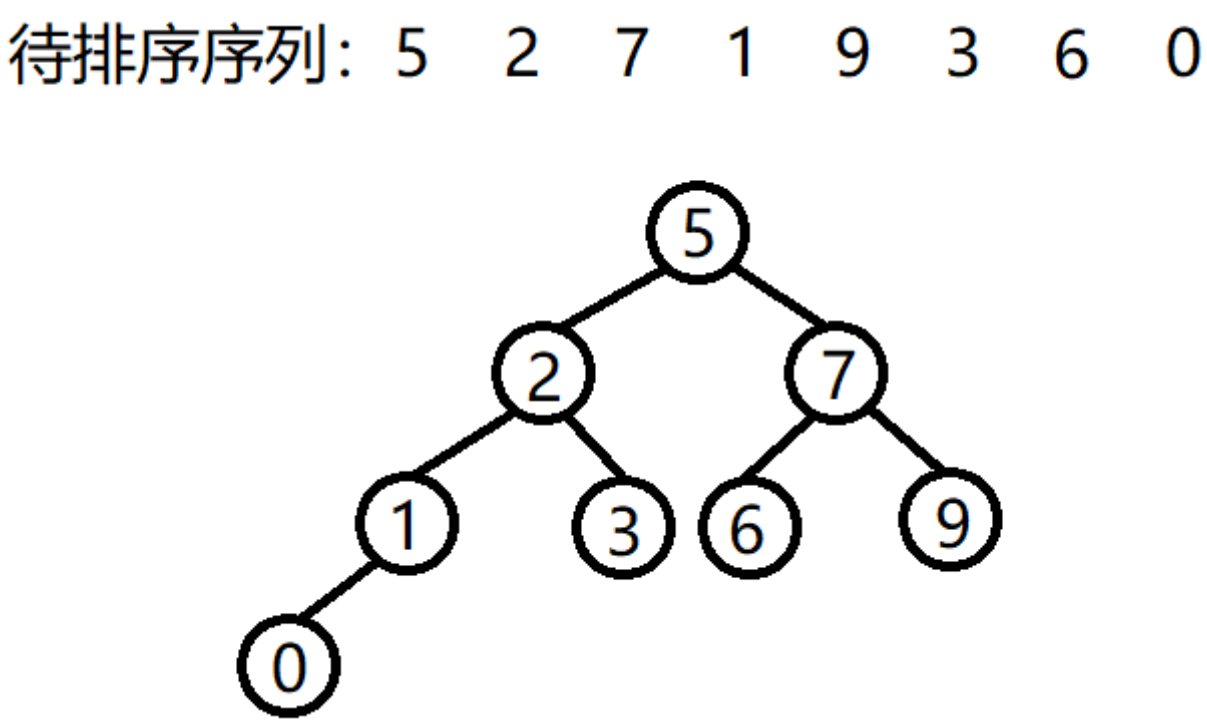
1.升序二叉排序树

在升序二叉排序树中，所有的节点都遵从这样的规则：左孩子 < 根节点 < 右孩子

也就是说：在一个新元素加入二叉排序树中的时候，如果比遇见的节点取值小，则向左走，直到成为子节点为止；

如果比遇见的节点取值大，则向右走，直到成为子节点为止

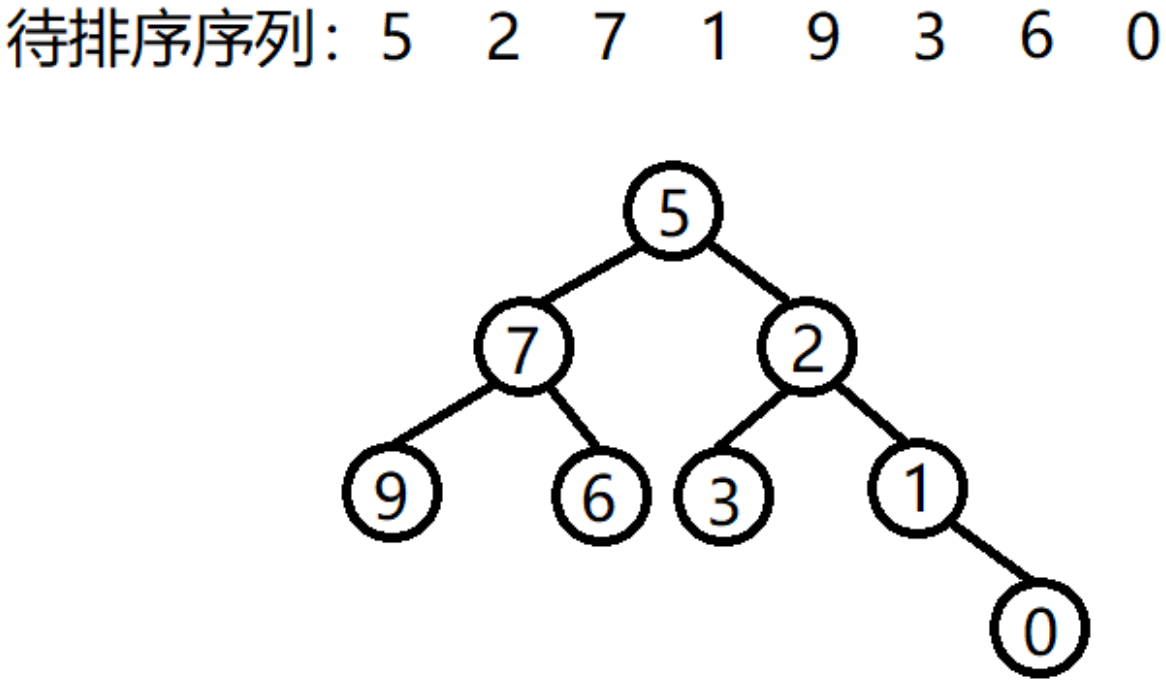
下面的图示展示的就是一个升序二叉排序树：



2.降序二叉排序树

降序二叉排序树中元素的顺序正好与升序二叉排序树中元素的存储顺序相反，他们是：左孩子 > 根节点 > 右孩子

那么上面的一组数据就会产生如下结构的二叉排序树：

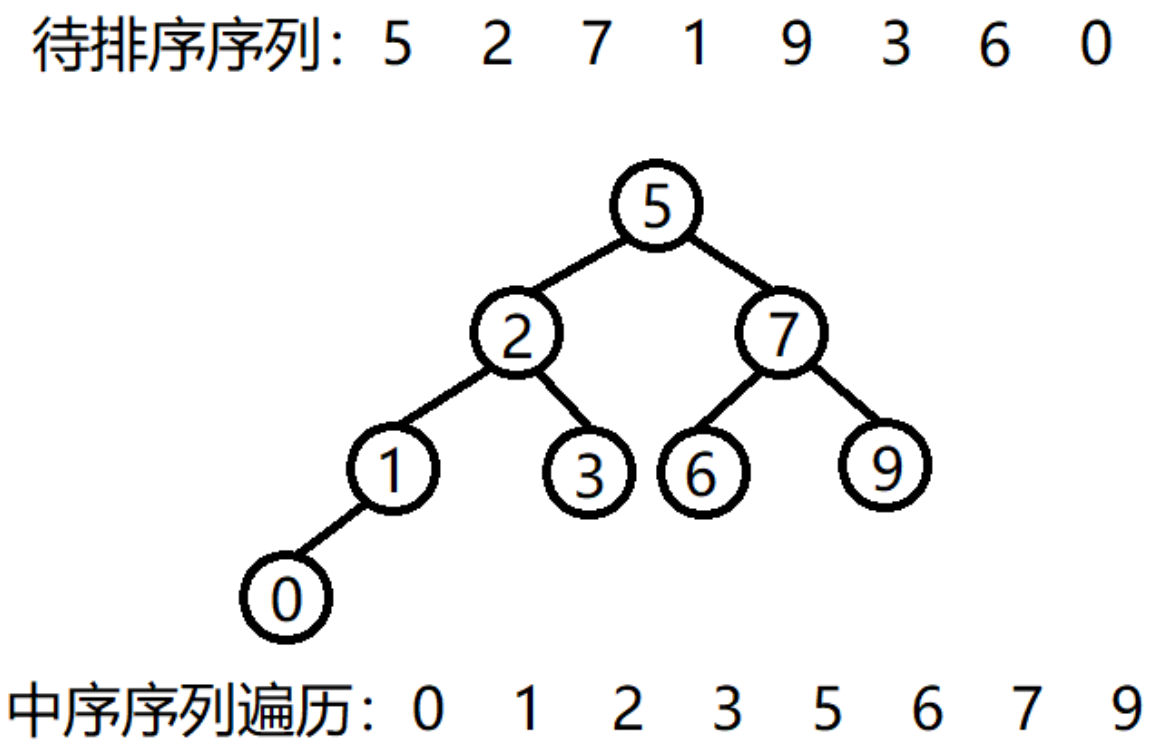


③得到有序序列：中序遍历二叉排序树

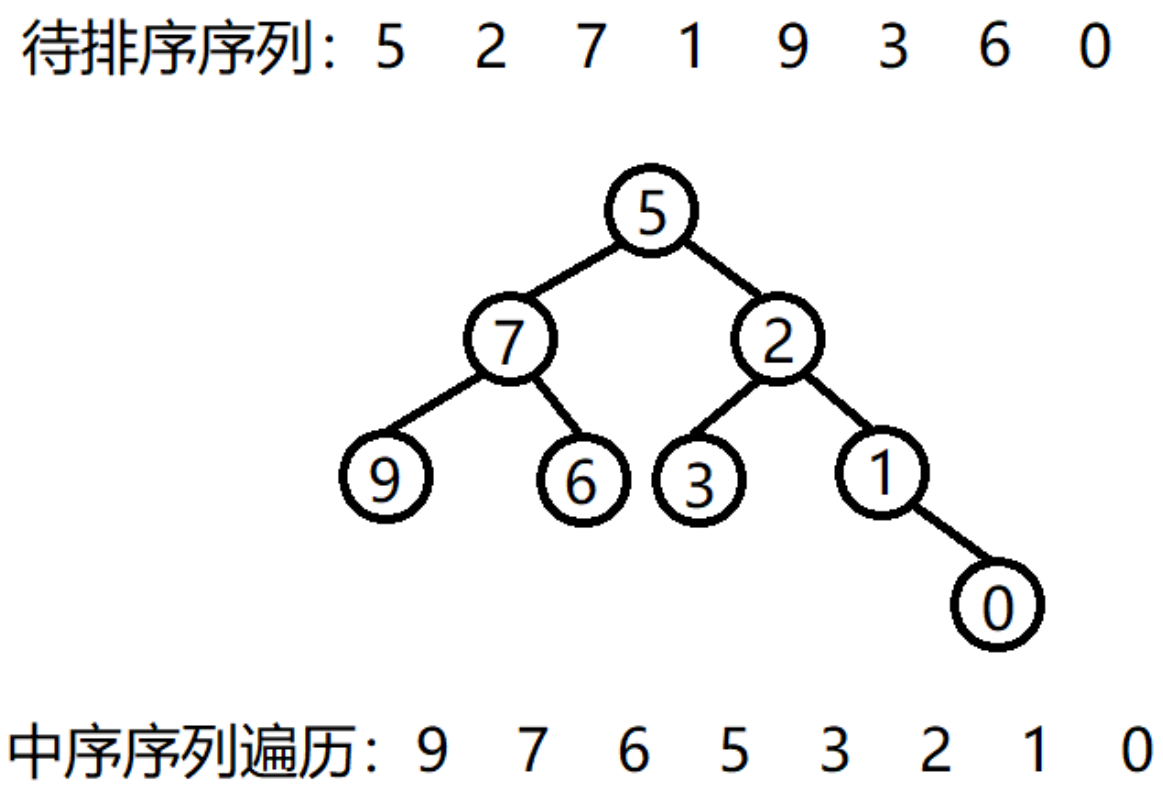
上面我们已经了解了二叉排序树的升序与降序的结构，那么二叉排序树是如何进行排序的呢?

我们不妨对上面的两棵二叉树分别进行中序序列遍历试试看：

升序二叉树：



降序二叉树：



我的天哪！太神奇了！竟然有序了呢！

没错，就是这么神奇！只要我们对二叉排序树进行中序序列遍历，就能够得到其中元素的有序序列。

④二叉排序树删除节点的操作

在删除二叉排序树中节点的时候，我们需要考虑如下几种情况：

情况1：删除的节点没有孩子节点

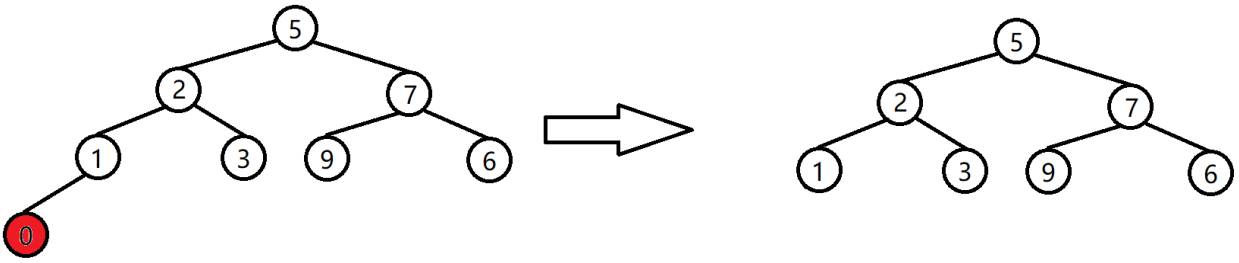
情况2：删除的节点只有左孩子或者右孩子

情况3：删除的节点同时具有左右孩子

我们接下来分别讨论这3种情况

1.删除的节点没有孩子节点

如果删除的节点没有孩子节点，那么直接删除这个节点就行了，不需要进行其他操作

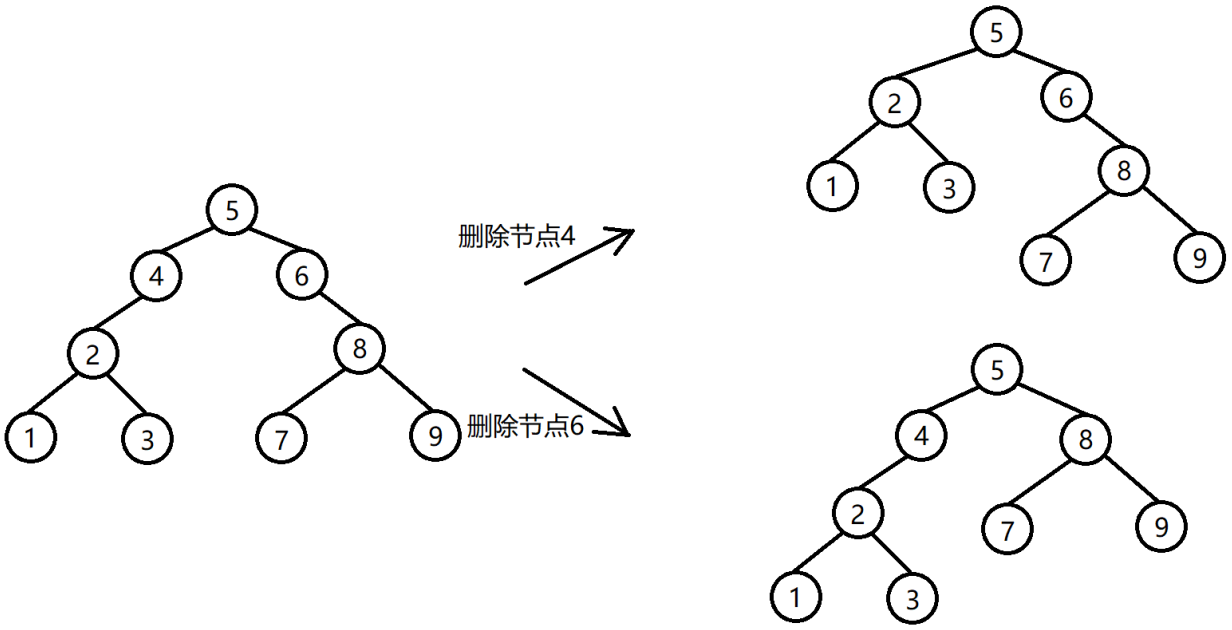


2.删除的节点只具有左孩子或者右孩子

如果删除节点下，只具有左孩子或者右孩子

那么在删除这个节点的时候，只要将其左孩子或者右孩子上移，替换这个节点的位置即可

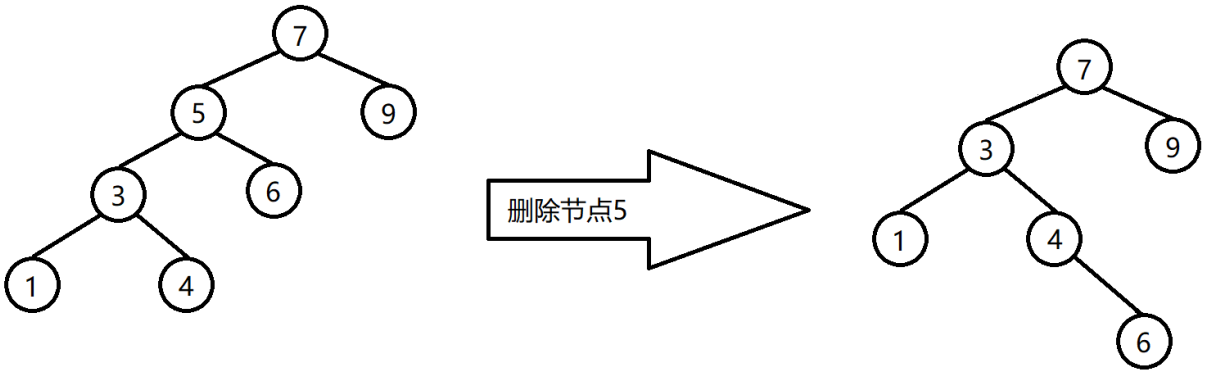
与父节点的相对位置保持不变



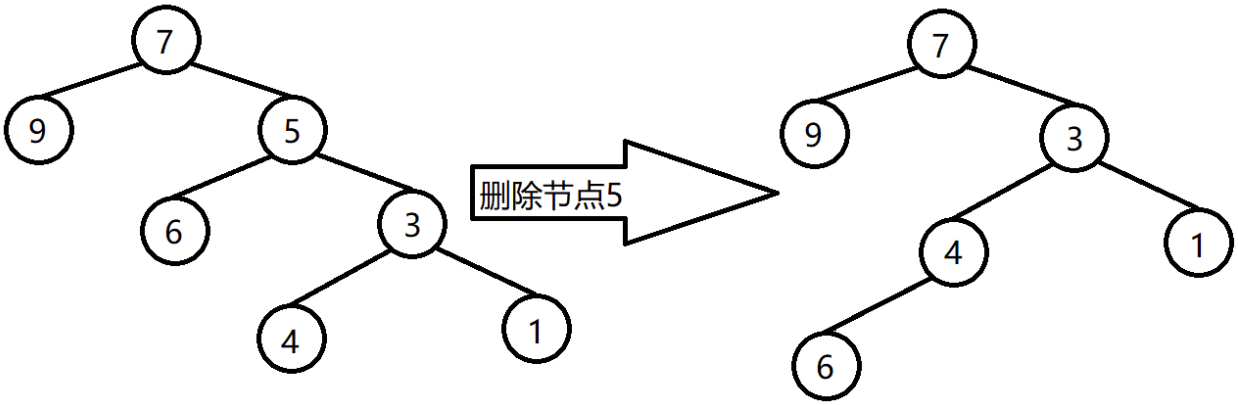
3.删除的节点同时具有左孩子和右孩子

在删除这样的节点的时候，我们需要考虑这个二叉排序树是升序结构还是降序结构：

在升序二叉树排序树中，删除这个节点之后，我们需要使用其左孩子向上提升，替代原来节点的位置，原来节点的右孩子，变成替代节点的右孩子



在降序二叉树排序树中，删除这个节点之后，我们需要使用其右孩子向上提升，替代原来节点的位置，原来节点的左孩子，变成替代节点的左孩子



经过上述删除操作之后二叉排序树中的节点，经过中序序列遍历最终依然是有序的

⑤逼死强迫症：二叉排序树退化为单链表

实际上，在一些特殊情况下，二叉排序树将会退化成为一种类似于单链表的结构

这种情况就是待排序序列完全正序或者完全倒序的情况

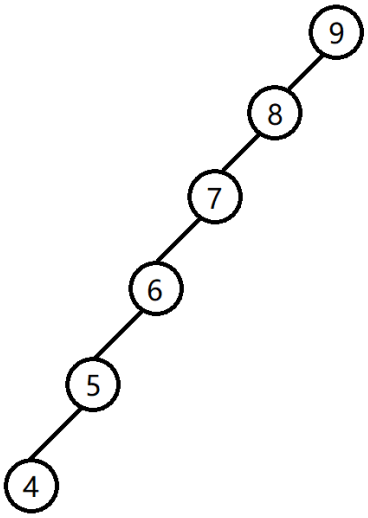
在这种情况下，所有的元素会按照加入二叉排序树的顺序，依次按照左孩子或者右孩子的方向逐个添加下去

也就是说，这个时候的二叉排序树只有左子树或者只有右子树，一条线下去……

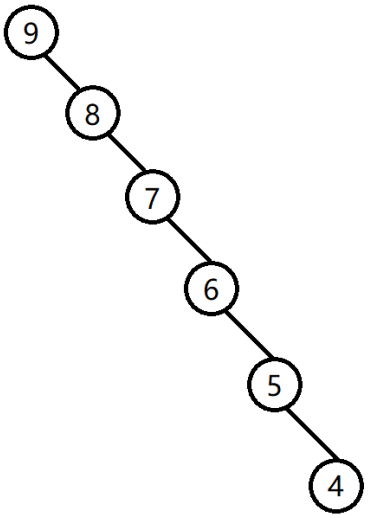
这样，一个二叉排序树就退化成为了单链表，其查找元素的速度大打折扣

待排序序列：9　8　7　6　5　4

升序二叉排序树



降序二叉排序树



在这种结构下，二叉排序树能够快速查找元素的优势完全发挥不出来，此时和使用单链表对元素进行有序排列是一样的，查找效率极低

那么这种情况要如何进行处理，才能够保证即使是完全有序的序列，也能够使用二叉排序树结构进行存储呢？

答案是使用平衡二叉树结构