超能二叉搜索树

今日目标：

1：能够说出二叉搜索树的定义及特点

2：能够手动实现一棵二叉搜索树，完成插入，查找，删除等操作

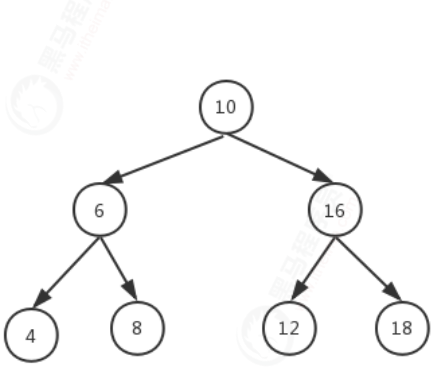
3：能够分析出二叉搜索树相关操作的时间复杂度

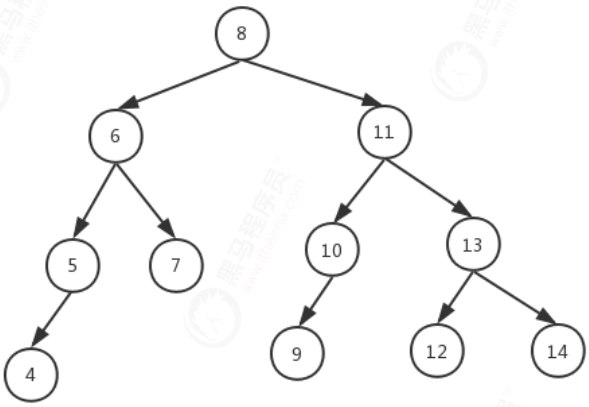
4：完成相关实战题目

1、定义及特点

二叉搜索树又名二叉查找树，有序二叉树或者排序二叉树，是二叉树中比较常用的一种类型。我们来看其定义：

二叉查找树要求，在树中的任意一个节点，其左子树中的每个节点的值，都要小于这个节点的值，而右子树节点的值都大于这个节点的值，





详细可以分为以下4点：

1. **若任意节点的左子树不空，则左子树上所有节点的值均小于它的根节点的值；**
2. **若任意节点的右子树不空，则右子树上所有节点的值均大于它的根节点的值；**
3. **任意节点的左、右子树也分别为二叉查找树；**
4. **没有键值相等的节点。**

另外：**对于二叉查找树而言，它的中序遍历结果是一个递增的序列！**

> 二叉搜索树有何特点?

我们从二叉搜索树这个名称就能够体会到该树的一个特点，能够支持快速查找，除此之外，二叉查找树支持动态数据的快速插入，删除，查找操作。这是它的一个操作特点，也正是我们要来学习它的原因。

之前也学习过散列表这种数据结构，它也支持这几个操作，并且散列表实现这几个操作更加的高效，时间复杂度是O(1),那既然有了如此高效的散列表为什么还要来学习二叉查找树，是不是有某些情况我们必须使用它？带着这几个问题我们依次展开二叉搜索树这几个特点的相关学习。

2、二叉搜索树的实现

想要深刻的了解二叉搜索树的相关特征，我们先来动手实现一个二叉搜索树。

创建二叉搜索树`com.itheima.tree.BinarySearchTree`，并定义树的节点对象

```
public static class TreeNode{
    int val;
    TreeNode left;
    TreeNode right;

    public int getVal(){
        return this.val;
    }
    public TreeNode(int val){
        this.val = val;
    }
    public TreeNode(TreeNode left,int val,TreeNode right){
        this.left = left;
        this.val = val;
        this.right = right;
    }
}
```

定义整棵树的根几点`root`

```
private TreeNode root;//根节点
```

重写`toString`方法，输出该二叉搜索树的中序遍历结果

```
@Override
public String toString() {
    StringBuilder sb = new StringBuilder();
    inOrder(root,sb);
    return sb.toString();
}

private void inOrder(TreeNode node,StringBuilder sb){
    if (node == null) {
        return;
    }
    inOrder(node.left,sb);
```

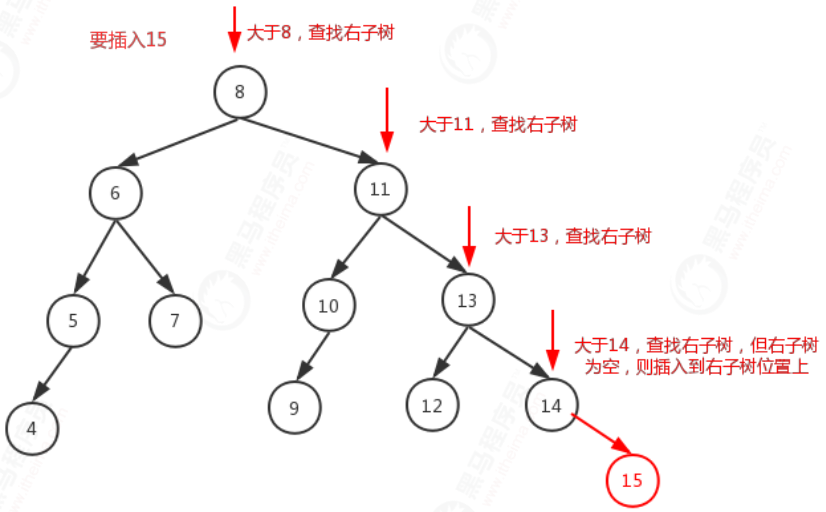
```
13      sb.append(node.val).append("->");
14      inOrder(node.right,sb);
15  }
```

2.1、插入操作

二叉搜索树的插入操作要先从根节点开始依次比较要插入的数据和节点的数据的大小并以此来判断是将数据插入其左子树还是右子树。

如果要插入的数据比节点数值大，并且节点的右子树为空，就将新数据直接插到右子节点的位置；如果不为空，就再递归遍历右子树，查找插入位置。

同理，如果要插入的数据比节点数值小，并且节点的左子树为空，就将新数据插入到左子节点的位置；如果不为空，就再递归遍历左子树，查找插入位置。



> https://visualgo.net/zh/bst

编写插入方法`add`

```
1   //添加
2   public boolean add(int val) {
3
4       if (root == null) {
5           root = new TreeNode(val);
6           return true;
7       }
8       //定义当前节点
9       TreeNode cur = root;
10      while (cur!=null) {
11          if (val > cur.val) { //添加到cur的右子树上
12              if (cur.right == null) {
```

```
                cur.right = new TreeNode(val);
                return true;
            }
            cur = cur.right;
        }else if (val < cur.val) { //添加到cur的左子树上
            if (cur.left == null) {
                cur.left = new TreeNode(val);
                return true;
            }
            cur = cur.left;
        }else {
            //可以更新,可以不做操作
            return false;
        }
    }
    return false;
}
```

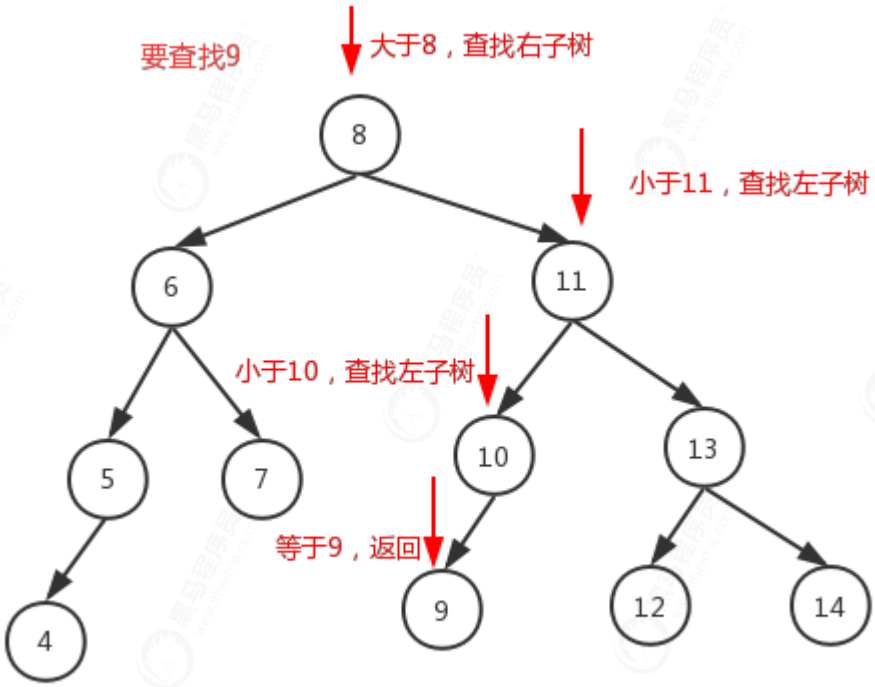
测试：创建测试类：`com.itheima.tree.BST_Test`

```
public class BST_Test {
    public static void main(String[] args) {
        BinarySearchTree bst = new BinarySearchTree();
        bst.add(10);
        bst.add(2);
        bst.add(4);
        bst.add(6);
        bst.add(10);
        bst.add(15);
        bst.add(16);
        bst.add(17);
        bst.add(18);
        bst.add(1);
        bst.add(3);
        bst.add(5);
        bst.add(9);
        bst.add(10);
        bst.add(11);
        bst.add(12);
        bst.add(13);
        bst.add(4);
        bst.add(7);
        System.out.println(bst);
    }
}
```

2.2、查找操作

我们来看如何在一棵二叉搜索树中查询某一个值的节点?

我们从根节点开始，如果它等于我们要查找的数据，那就返回。如果要查找的数据比根节点的值小，那就在左子树中递归查找；如果要查找的数据比根节点的值大，那就在右子树中递归查找。如图



编写根据值查找节点的方法`get`

```
//根据val查找节点
public TreeNode get(int val) {
    TreeNode cur = root;
    while (cur != null) {
        if (val > cur.val) { //去右子树查找
            cur = cur.right;
        }else if (val < cur.val) { //去左子树查找
            cur = cur.left;
        }else { //找到了
            return cur;
        }
    }
    return cur;
}
```

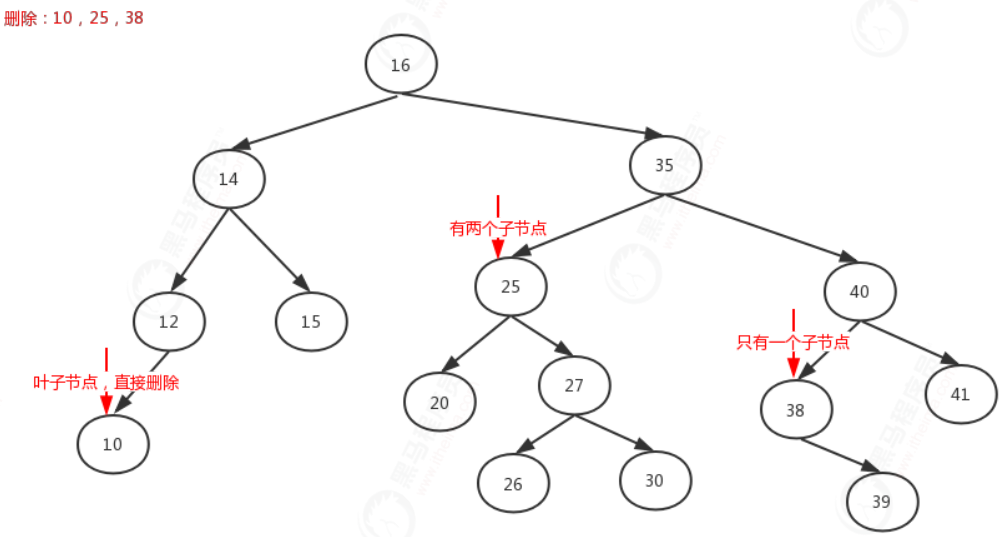
> 思考：有人可能会有疑问，这里每个节点中除了存储值之外就只有两个指针left和right了，我都已经知道值了，还根据值查找节点TreeNode有什么作用呢？
>
> 实际工程应用中TreeNode中可以根据需要存储很多数据，这里的值val就相当于一个ID，节点中存储ID和对应的业务数据，查找的时候根据ID查找业务数据。
>
> 或者类比HashMap，它的节点中就存储了key，hash值，Value；相关操作都是根据key和hash值来操作的，获取的是存储在节点中存储的Value

测试：添加测试代码如下

```
System.out.println(bst.get(10));
```

2.3、删除操作

二叉搜索树的插入和查询相对来说比较简单易懂，但是删除操作相对复杂，总结下来有三种情况：

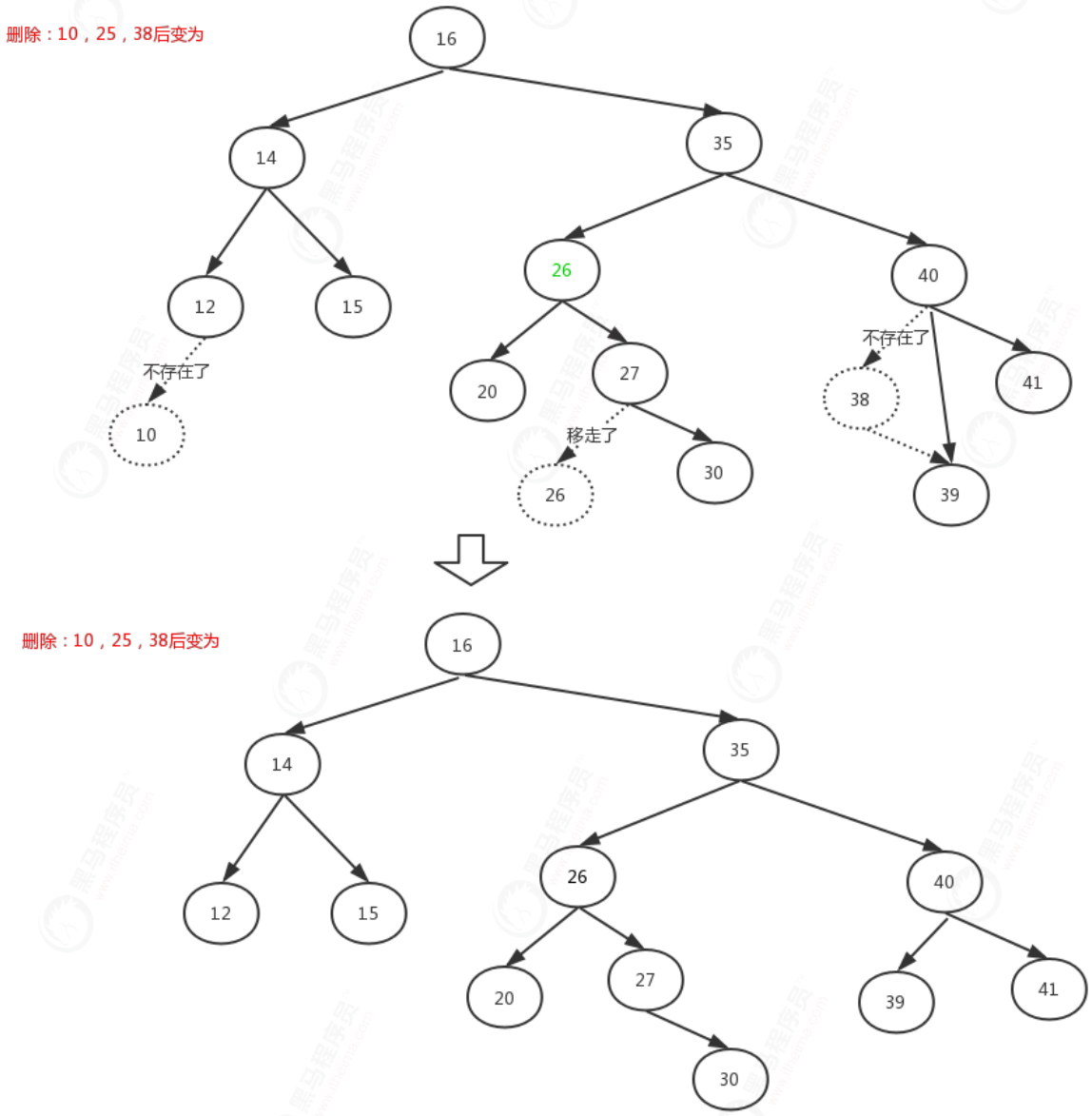


1：要删除的节点是叶子节点即没有子节点，我们只需将父节点中指向该节点的指针置为null即可，这是最简单的一种形式。比如删除图中的节点10

2：要删除的节点只有一个子节点(只有左子节点或者只有右子节点)，我们只需要更新父节点中，指向要删除节点的指针，让它指向要删除节点的子节点就可以了。比如删除图中的节点38

3：要删除的节点有两个子节点，这是最复杂的一种情况，我们需要找到这个节点的右子树中的最小节点，把它替换到要删除的节点上。然后再删除掉这个最小节点，因为最小节点肯定没有左子节点（如果有左子结点，那就不是最小节点了），所以，我们可以应用上面两条规则来删除这个最小节点。比如删除图中的节点25

按照规则删除之后的树结构为：



编写删除方法`remove`

```
//根据值val删除节点
public void remove(int val) {
    //定义要删除的节点 del
    TreeNode del = root;
    //定义要删除节点的父节点del_p
    TreeNode del_p = null;

    while (del != null) {
        if (val > del.val) { //从右子树中找要删除的节点
            del_p = del;
            del = del.right;

        }else if (val < del.val) {//从左子树中找要删除的节点
```

```
            del_p = del;
            del = del.left;
        }else {//找到了要删除的节点
            break;
        }
    }
    //如果没有找到要删除的节点则直接返回
    if (del == null) {
        return;
    }

    //判断不同的情况

    //情况1：如果要删除的节点有两个子节点
    if ( del.left != null && del.right !=null) {
        /**
             * 将当前要删除节点的值用右子树中最小节点的值替换,最后把右子树中最小节点删除
             * 而删除右子树中最小节点我们可以将del指针指向该右子树最小节点,同时del_p指向它的父节点
             * 这样删除操作就可以和情况2合并了
             */
        //从右子树中找最小节点及其父节点----其实就是找最左侧的叶子节点和其父节点
        TreeNode min = del.right;
        TreeNode min_p = null;

        while (min.left != null) {
            min_p = min;
            min = min.left;
        }
        //用最小节点的值替换当前要删除的节点的值
        del.val = min.val;
        //将del指向min,del_p指向min_p;方便删除最小节点
        del = min;
        del_p = min_p;

    }

    //情况2：如果要删除的节点有一个子节点(不论左子节点还是右子节点)或者它本身是叶子节点 我们需要找到它的子节点

    TreeNode del_child = null;

    if (del.right != null) {
        del_child = del.right;
    }else if (del.left != null) {
        del_child = del.left;
    }


    //最后执行删除,将del_p指向del的指针指向del_child,
    if (del_p.left == del) {
        del_p.left = del_child;
    }else if (del_p.right == del) {

        del_p.right = del_child;
```

```
    }
    //同时清空del指向del_child的指针
    del.left = null;
    del.right = null;
}
```

测试：添加测试代码

```
bst.remove(10);
System.out.println(bst);
bst.remove(18);
System.out.println(bst);
bst.remove(12);
System.out.println(bst);
bst.remove(1);
System.out.println(bst);
```

查看输出

2.4、查找最值

对于查找二叉查找树的最小值，我们只需要从根节点开始依次查找其左子节点直到最后的叶子节点，最后的叶子节点就是其最小值，同理查找最大值只需要从根节点开始依次查找其右子节点直到最后的叶子节点即为最大值。

代码实现如下：

```
//获取BST中的最大值节点
public TreeNode getMax(){
    //根节点右子树中的最后侧叶子节点
    if (root == null) {
        return null;
    }
    TreeNode max = root;
    while (max.right != null) {
        max = max.right;
    }
    return max;
}

//获取BST中的最小值节点
public TreeNode getMin(){
    //根节点左子树中的最左侧叶子节点
    if (root == null) {
        return null;
    }
    TreeNode min = root;
    while (min.left != null) {
        min = min.left;
    }
    return min;
```

```
25  }
```

> 测试：添加测试代码

```
1  System.out.println("最大节点值:"+bst.getMax().getVal());
2  System.out.println("最小节点值:"+bst.getMin().getVal());
```

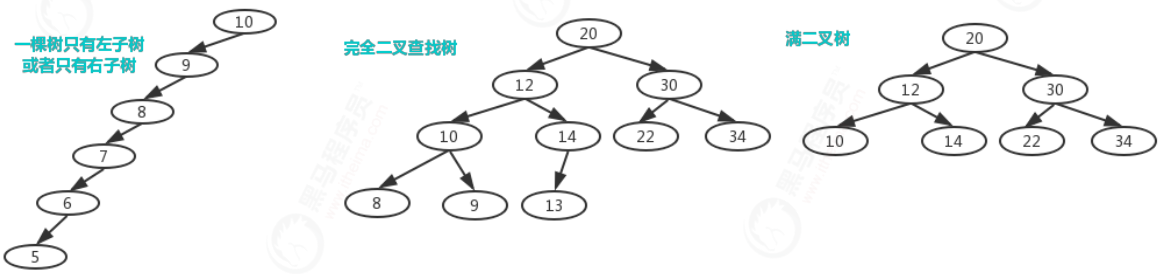
> 查看输出

> 课后作业：
>
> 对于二叉搜索树我们还可以找到节点的前驱节点和后继节点，对于这两个概念我们解释说明如下：
>
> **后继节点：该节点右子树中最小的节点**
>
> **前驱节点：该节点左子树中最大的节点**

2.5、时间复杂度分析

我们分析一下二叉查找树的查找，插入，删除的相关操作的时间复杂度。

实际上由于二叉查找树的形态各异，时间复杂度也不尽相同，我画了几棵树我们来看一下插入，查找，删除的时间复杂度



对于图中第一种情况属于最坏的情况，二叉查找树已经退化成了链表，左右子树极度不平衡，此时查找的时间复杂度肯定是O(n)。

对于图中第二种或者第三种情况是属于一个比较理想的情况，我们代码的实现逻辑以及图中所示表明**插入，查找，删除的时间复杂度其实和树的高度成正比，那也就是说时间复杂度为O(height)**

> 那如何求一棵完全二叉树的高度？即求一棵包含n个节点的完全二叉树的高度？
>
> 对于一棵满二叉树而言：树的高度就等于最大层数减一，为了方便计算，我们转换成层来表示。从上图中可以看出，包含 n 个节点的完全二叉树中，第一层包含 1 个节点，第二层包含 2 个节点，第三层包含 4 个节点，依次类推，下面一层节点个数是上一层的 2 倍，第 K 层包含的节点个数就是 `2^(k-1)`。
>
> 但是对于完全二叉树来说，最后一层的节点个数有点儿不遵守上面的规律了。它包含的节点个数在 1 个到 `2^(k-1)` 个之间（我们假设最大层数是 k）。如果我们把每一层的节点个数加起来就是总的节点个数 n。也就是说，如果节点的个数是 n，那么 n 满足这样一个关系：

```
1+2+4+8+...+2^(k-2) +1 =<n <= 1+2+4+8+...+2^(k-2) +2^(k-1)
```

这是一个等比数列，根据等比数列求和公式S= a1(1-`q^(n-1)`)/ 1-q，其中q是公比，n为数据个数。

所以：我们利用求和公式对上述式子进行计算后得知，k的一个最大值是：

$$log_2 n + 1$$

也就是说完全二叉树的高度小于等于：

$$log_2 n$$

通过我们的分析我们发现一棵极度不平衡的二叉查找树，它的查找性能和单链表一样。我们需要构建一种不管怎么删除、插入数据，在任何时候，都能保持任意节点左右子树都比较平衡的二叉查找树，这种特殊的二叉查找树也可以叫做**平衡二叉查找树。平衡二叉查找树的高度接近logn，所以插入、删除、查找操作的时间复杂度也比较稳定，是 O(logn)。**

2.6、对比散列表

之前我们学习过散列表的插入、删除、查找操作的时间复杂度可以做到常量级的 O(1)，非常高效。

而二叉搜索树在比较平衡的情况下，插入、删除、查找操作时间复杂度才是 O(logn)，相对散列表，好像并没有什么优势，那我们为什么还要用二叉查找树呢？原因有如下几点：

1：散列表中的数据是无序存储的，如果要输出有序的数据，需要先进行排序。而对于二叉查找树来说，我们只需要中序遍历，就可以在 O(n) 的时间复杂度内，输出有序的数据序列。

2：散列表扩容耗时很多，而且当遇到散列冲突时，性能不稳定，尽管二叉查找树的性能也不稳定，但是在工程中，我们最常用的平衡二叉查找树的性能非常稳定，时间复杂度稳定在 O(logn)。

3：尽管散列表的查找等操作的时间复杂度是常量级的，但因为哈希冲突的存在，这个常量不一定比 logn 小，所以实际的查找速度可能不一定比 O(logn) 快。加上哈希函数的耗时，也不一定就比平衡二叉查找树的效率高。

4：散列表的构造比二叉查找树要复杂，需要考虑的东西很多。比如散列函数的设计、冲突解决办法、扩容、缩容等。平衡二叉查找树只需要考虑平衡性这一个问题，而且这个问题的解决方案比较成熟、固定。

5：为了避免过多的散列冲突，散列表装载因子不能太大，特别是基于开放寻址法解决冲突的散 列表，不然会浪费一定的存储空间。

综合这几点，平衡二叉查找树在某些方面还是优于散列表的，所以，这两者的存在并不冲突。我们在实际的开发过程中，需要结合具体的需求来选择。

3、实战题目

96. 不同的二叉搜索树

字节，Facebook面试题，96. 不同的二叉搜索树

1：按照 BST 的定义，如果整数 1 到 n 中的整数 k 作为根节点值，则 1 ~ k-1 会去构建左子树，k+1 ~ n 会去构建右子树。

2：以 k 为根节点的 BST 种类数 = 左子树 BST 种类数 * 右子树 BST 种类数。

3：最后问题变成：计算不同的 k 之下，上面等式右边的种类数的累加结果。

解题思路：递归

初始代码：

```
class Solution {
    public int numTrees(int n) {
       if (n <=1) {
            return 1;
        }
        int sum = 0;
        for (int i=1;i<=n;i++) { //计算不同的 k 之下[1,k-1],[k+1,n]构成不同BST种类乘积的累加
            sum += numTrees(i-1) * numTrees(n-i);
        }
        return sum;
    }
}
```

> 时间太慢，为什么？有很多重复计算，如何解决？

记忆化递归：

```
class Solution {
    public int numTrees(int n) {
        return recur(n,new HashMap());
    }

    public int recur(int n,Map<Integer,Integer> map) {
        if (n <=1) {
            map.put(n,1);
            return 1;
        }
        if (map.containsKey(n)) {
            return map.get(n);
        }
        int sum = 0;
        for (int i=1;i<=n;i++) { //计算不同的 k 之下[1,k-1],[k+1,n]构成不同BST种类乘积的累加
            sum += recur(i-1,map) * recur(n-i,map);
        }
        map.put(n,sum);
        return sum;
    }

}
```

95. 不同的二叉搜索树 II

亚马逊，字节半年内面试题，95. 不同的二叉搜索树 II

跟上一题有一点不一样的是，上一题只需要求出种类数量就可以了，这道题目是需要求出每一种可能的结果

```
class Solution {
    public List<TreeNode> generateTrees(int n) {
        if ( n < 1) {
            return new ArrayList();
        }
        return recurGen(1,n);
    }

    //根据数据区间生成该区间内所有的BST
    public List<TreeNode> recurGen(int start,int end) {
        List<TreeNode> res = new ArrayList();
        if (start>end) {
            res.add(null);
            return res;
        }

        //在[start,end]区间内，依次以i为根节点,[start,i-1]为左子树区间,[i+1,end]为右子树区间构造BST
        for (int i=start;i<= end;i++) {
            List<TreeNode> lefts = recurGen(start,i-1);//所有左子树集合
            List<TreeNode> rights = recurGen(i+1,end);//所有右子树集合

            //从左子树集合和右子树集合中分别选一种组合，结合当前根节点i构造BST
            for (TreeNode left:lefts) {
                for (TreeNode right : rights) {
                    TreeNode root = new TreeNode(i);
                    root.left = left;
                    root.right = right;

                    //将组合好的BST添加到最终集合
                    res.add(root);
                }
            }
        }
        return res;
    }

}
```

108. 将有序数组转换为二叉搜索树

亚马逊，字节，苹果最近面试题，108. 将有序数组转换为二叉搜索树

递归解法

```java
class Solution {
    public TreeNode sortedArrayToBST(int[] nums) {
        return recur(nums,0,nums.length-1);
    }

    public TreeNode recur(int[] nums,int start,int end){
        if (start > end) {
            return null;
        }
        //选择该区间的中点作为根节点
        int mid = (end -start) /2 + start;
        TreeNode root = new TreeNode(nums[mid]);
        //重复子问题构造左子树，右子树
        root.left = recur(nums,start,mid-1);
        root.right = recur(nums,mid+1,end);
        return root;
    }
}
```

> 查阅一下官方题解！

144. 二叉树的前序遍历

二叉树的前，中，后序遍历用递归算法写起来非常简单，而使用迭代算法完成前，中，后序遍历才是本题的难点，其本质是在模拟递归，因为在递归的过程中使用了系统栈，所以在迭代的解法中常用`Stack`来模拟系统栈。

```java
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        if (root ==null) {
            return new ArrayList();
        }
        Stack<TreeNode> stack = new Stack();
        stack.push(root);
        List<Integer> res = new ArrayList();
        while (!stack.isEmpty()) {
            TreeNode pop = stack.pop();
            res.add(pop.val);

            if (pop.right!=null) {
                stack.push(pop.right);
            }
            if (pop.left !=null) {
                stack.push(pop.left);
            }
        }

        return res;
```

```
21      }
22  }
```

同样的，可以用迭代算法完成其他两种形式的遍历！

1. 94. 二叉树的中序遍历
2. 145. 二叉树的后序遍历