二叉搜索树和平衡树

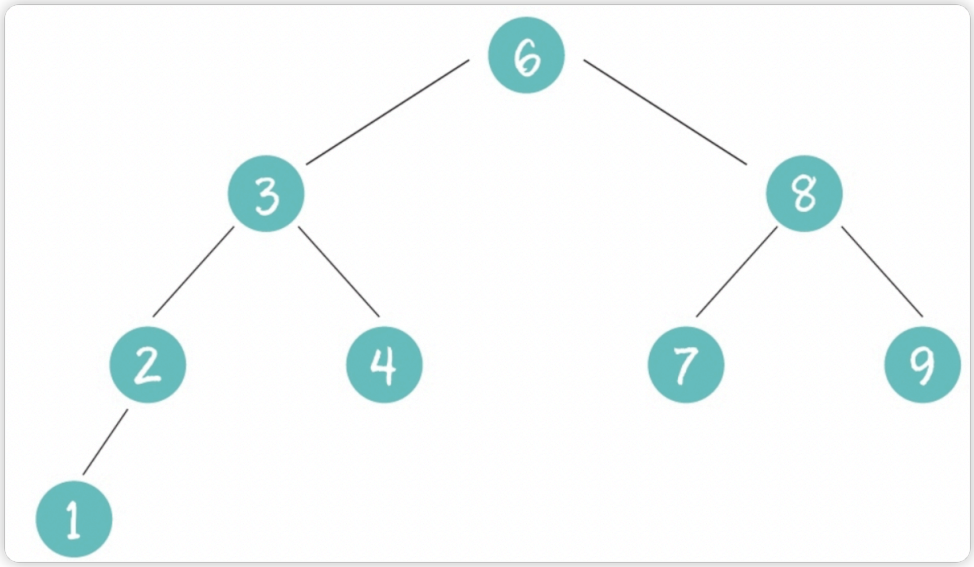
1. 二叉搜索树（BST）

二叉搜索树(Binary Search Tree)，也有称之为二叉排序树、二叉查找树

1.1 定义和性质

在二叉树的基础上，增加了几个规则约束：

- 如果他的左子树不空，则左子树上所有结点的值均小于它的根结点的值。
- 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值。
- 它的左、右树又分为二叉排序树。



1.2 构建二叉排序树

> 假设有无序序列如下：
>
> 8 3 10 1 6 14 4 7 13

1.3 二叉搜索树的优势

- 二叉排序树的中序遍历，就是一个从小到大排好序的序列，但是查找时，完全没有必要先进行中序遍历生成一个有序的数组，再二分查找，直接根据二叉搜索树的约束直接操作。
- 查找时间最坏情况就是树的深度
- 二叉搜索树的查找逻辑，可以写成递归的思路和非递归思路。

```
BiTree SearchBST(BiTree T,KeyType key){
    //如果递归过程中 T 为空，则查找结果，返回NULL；或者查找成功，返回指向该关键字的指针
    if (!T || key==T->data) {
        return T;
    }else if(key<T->data){
        //递归遍历其左孩子
        return SearchBST(T->lchild, key);
    }else{
```

```
        //递归遍历其右孩子
        return SearchBST(T->rchild, key);
    }
}
```

1.4 二叉搜索树的实现

1.4.1 二叉搜索树的插入操作

- 算法思想
    - 插入的新节点都是在叶节点位置
    - 寻找插入节点的位置，建立父节点和新节点的左右关系
    - 是一种递归思想
- 递归思路
    - 涉及到前后节点，子问题就是返回新节点，上一个状态就是左子树还是右子树来接收
- 递归核心代码

1.4.2 查找某一节点下的最大或最小值

- 算法思想
    - 比当前节点小的值，一定放在他的左子树上，那么一直往左查找最左边的节点，就是最小值
    - 比当前节点大的值，一定放在他的右子树上，那么一直往右查找最左边的节点，就是最大值
- 算法实现

1.4.2 二叉搜索树的删除操作

- 算法分析
    - 删除节点，共有三种可能性：

叶子结点

- 叶子结点从二叉搜索树中移除后，并不影响其他结点的排列规则，直接删除

度为1的结点

- 该节点缺失左子树或右子树，当去掉这个节点后，剩余的左子树和右子树满足二叉搜索树的要求
- 矛盾在于删除的这个节点，属于父节点的左边还是右边，那么剩余的左子树或右子树仍然满足父节点的左右属性，接入这个父节点就可以

度为2的结点

- 假设要删除的节点A，他的后结点，既有大于他的也有小于他的，现在就是要选择一个节点B，满足左边节点都比B小，右边节点都比B大。
- 其实就是中序遍历时，前一个节点或后一个节点来替换这个节点，然后删除前一个或后一个节点

先找到这个待删除结点的前一个或后一个，替换待删除结点

1234567

1234457

这个时候删除4就可以，而4一定是叶子结点，就又回到了叶子节点删除的逻辑

- 递归思路
- 二叉搜索树的删除请情况
    - 删除叶子节点，直接删除
    - 删除的节点有一个叶子节点，用叫子节点来替代
    - 删除的节点有两个子节点

- 找到前驱节点，复制前驱节点的值覆盖掉预备删除的节点，然后删除前驱节点
- 找到后继节点，复制后继节点的值覆盖掉预备删除的节点，然后删除后继节点

2. 二叉平衡树（AVL）

> AVL树是最早被发明的自平衡二叉查找树。在AVL树中，任一节点对应的两棵子树的最大高度差的绝对值为1，因此它也被称为高度平衡树。AVL树是根据它的发明者G.M. Adelson-Velsky和E.M. Landis命名的。
>
> 查找、插入和删除在平均和最坏情况下的时间复杂度都是 O(logn） 。 增加和删除元素的操作则可能需要借由一次或多次树旋转，以实现树的重新平衡。

2.1 二叉搜索树的问题

- 二叉搜索树的查找效率取决于树的高度，因此保持树的高度最小，即可保证树的查找效率。
- 二叉搜索树会退化成单链表，搜索效率降低为O(n)

> 二叉搜索树一定程度上可以提高搜索效率，但是当原序列有序时，例如序列 A = {1，2，3，4， 5，6}，构造二叉搜索树如图。依据此序列构造的二叉搜索树为右斜树，同时二叉树退化成单链表，搜索效率降低为 O(n)。
>
> 二叉搜索树的查找效率取决于树的高度，因此保持树的高度最小，即可保证树的查找效率。

2.2 平衡因子

> 左子树与右子树的高度差即为该节点的平衡因子（BF,Balance Factor）
>
> 平衡二叉树中不存在平衡因子大于 1 的节点。
>
> 在一棵平衡二叉树中，节点的平衡因子只能取 0 、1 或者 -1 ，分别对应着左右子树等高，左子树比较高，右子树比较高。

2.3 平衡树失衡情况分析

2.3.1 左旋

- 当加入一个新节点后，右边的高度增加，导致失衡

```
/* 左旋操作
 *     px
 *     |
 *     x
 *   /  \
 * lx    y
 *      / \
 *     ly ry
 * */
AVLNode *leftRotate(AVLNode *x);
```

2.3.2 右旋

```
/* 右旋操作
 *       py
 *       |
 *       y
 *     /  \
 *    x   ry
```

```
 *   / \
 * lx  rx
 * */
AVLNode *rightRotate(AVLNode *y);
```

2.3.3 LL、RR、LR、RL

- LL ： 失衡节点左边高，新插入节点是在失衡节点左孩子的左边。直接对失衡节点进行右旋即可。
- RR：失衡节点右边高，新插入节点是在失衡节点右孩子的右边。直接对失衡节点进行左旋即可。
- LR：失衡节点左边高，新插入节点是在失衡节点左孩子的右边。先对左孩子进行左旋，再对失衡节点进行右旋。
- RL：失衡节点右边高，新插入节点是在失衡节点右孩子的左边。先对右孩子进行右旋，再对失衡节点进行左旋。

2.4 平衡二叉树的插入操作

- 算法思路
    - 先找到要插入的节点位置，插入新节点
    - 更新平衡因子，根据平衡因子小于-1和大于1的情况进行讨论

2.5平衡二叉树的删除操作

- 算法思路
    - 依据二叉排序树的删除特点，依然按照3种条件进行删除，保证最后删除的都是叶子节点即可。
    - 节点删除后，依次对节点进行平衡因子检查，若发现某节点不平衡，按照平衡调节法进行调节。