PS5-231830106

Problem 1

(a): 唯一性证明

- 数学归纳法：
  1. 归纳基础：若n=1，显然这棵树唯一
  2. 归纳假设：假设从n=1到n=k-1，这棵树都是唯一的
  3. 归纳步骤：若n=k时，因为后序遍历的最后一个节点为root，而root在中序遍历的序列中可以划分出左子树和右子树，显然这两颗树的节点数明显小于k，由归纳假设可知，左子树和右子树唯一，则显然整棵树唯一，证毕。

(b): 存在性证明

- 伪代码如下，构造思路类似于唯一性证明。

```
def buildTree(post_order, in_order):
    # 根节点是后序遍历的最后一个节点
    root_val = post_order[-1]
    root = TreeNode(root_val)

    # 在中序遍历中找到根节点的位置
    root_index_in_order = in_order.index(root_val)

    # 划分中序遍历的左子树和右子树
    left_in_order = in_order[:root_index_in_order]
    right_in_order = in_order[root_index_in_order + 1:]

    # 划分后序遍历的左子树和右子树
    left_post_order = post_order[:len(left_in_order)]
    right_post_order = post_order[len(left_in_order):-1]

    # 递归构造左子树和右子树
    root.left = buildTree(left_post_order, left_in_order)
    root.right = buildTree(right_post_order, right_in_order)

    return root
```

Problem 2

1. 思路：step1通过中序遍历将BST转化为一个升序排列的数组，再通过分治将数组构造成一个二叉平衡搜索树。
2. 伪代码

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# 中序遍历：将二叉搜索树转化为有序数组
def inorder_traversal(node, result):
    if not node:
        return
    inorder_traversal(node.left, result)
    result.append(node.val)
    inorder_traversal(node.right, result)

# 根据有序数组构造平衡的二叉搜索树
def sorted_array_to_bst(nums):
    if not nums:
        return None
    mid = len(nums) // 2
    root = TreeNode(nums[mid])
    root.left = sorted_array_to_bst(nums[:mid])
    root.right = sorted_array_to_bst(nums[mid+1:])
    return root

# 主函数：将 BST 转换为高度平衡的 BST
def balance_bst(root):
    sorted_nodes = []
    # 第一步：对 BST 进行中序遍历，获取有序数组
    inorder_traversal(root, sorted_nodes)
    # 第二步：使用有序数组重新构建平衡的 BST
    return sorted_array_to_bst(sorted_nodes)
```

3. 时间复杂度：中序遍历时间复杂度显然为$O(n)$，构建平衡二叉搜索树时对每个节点都只操作一次，因此时间复杂度也为$O(n)$，因此总时间复杂度为$O(n)$。

Problem 3

1. 思路：先中序遍历二叉树，并将遍历得到的数组排序，再将二叉树通过旋转变成一个chain，寻找数组中第k小的值，再通过左旋右旋和交换将第k小最小值移到最前面，即可

转化为BST。

2. 伪代码

```
# 定义一个节点类
class Node:
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None

# 主函数
def binary_tree_to_bst_with_kth_element(root, k):
    # Step 1: 中序遍历得到所有节点值
    values = []
    inorder_traversal(root, values)

    # Step 2: 对遍历结果排序
    values.sort()

    # Step 3: 通过旋转将二叉树转化为右链
    root = flatten_to_right_chain(root)

    # Step 6: 使用 changesort 进行调整
    root = changesort(root)

    return root

# 中序遍历二叉树，将节点值存入数组
def inorder_traversal(node, values):
    if node:
        inorder_traversal(node.left, values)
        values.append(node.value)
        inorder_traversal(node.right, values)

# 通过右旋将二叉树转化为右链
def flatten_to_right_chain(node):
    while node is not None:
        if node.left:
            node = right_rotate(node)
        else:
            node = node.right
    return node
```

```
# 将第 k 小的值移到链的最前面
def move_front(node):
    current = node
    x = current
    next = current.right
    y = current.right.right
    left_rotate(next)
    left_rotate(x)
    swap(x)
    right_rotate(y)
    right_rotate(x)

# 将chain排序
def change_sort(root):
        current = root
        for i in range(n):
                target = values[i]
                while current.right.values != target:
                        current = current.right
                move_to_front(current);
        return root;

# 右旋转操作
def right_rotate(node):

# 左旋转操作
def left_rotate(node):

# 交换操作
def swap(node):
```

3. 时间复杂度：中序遍历和排序不消耗操作的时间复杂度，操作复杂度消耗在：旋旋转为一个chain后，重复寻找第k大的元素并将其移到前面。旋转为chain大操作复杂度为$O(n)$，对每个元素都要进行5次操作，最后的操作复杂度为$O(n)$。

Problem 4

(a):

```
        88(B)

     88(B)
    /
```

```
   77(R)

      77(B)
     /   \
   66(R) 88(R)

      77(B)
     /   \
   66(B) 88(B)
   /
 22(R)

      77(B)
     /   \
   33(B) 88(B)
   /  \
 22(R) 66(R)

   33(B)
   /   \
  22(B) 77(B)
 /      / \
11(R) 66(R)88(R)
```

(b):

```
   33(B)
   /   \
  22(B) 77(B)
 /      / \
11(R) 66(R)88(R)

   33(B)
   /   \
  22(B) 77(B)
        / \
         66(R)88(R)


     77(B)
     /  \
   66(B) 88(B)
  /
```

```
33(R)

     77(B)
     /  \
   66(B) 88(B)

     77(B)
        \
        88(R)

     88(B)
```

Problem 5

(a):

不妨设$N(h)$为高度为h的AVL树的最小节点数，由AVL树的定义可以知左子树和右子树的节点差至多为一，可以写出递推式：
$N(h) = N(h-1) + N(h-2) + 1$
设F_h为斐波那契数列的第h项，可以注意到$N(h) = F_{h+2} - 1$，已知斐波那契数$F_h = \frac{1}{\sqrt{5}} \times \phi^h$，则有$n \geq \frac{1}{\sqrt{5}} \times \phi^h$，则$h = O(\log n)$。即证。

(b):

先定义左旋与右旋

```
# 右旋
def right_rotate(x):
    y = x.left
    x.left = y.right
    y.right = x
    x.h = max(height(x.left), height(x.right)) + 1
    y.h = max(height(y.left), height(y.right)) + 1
    return y  # y is the new root

# 左旋
def left_rotate(x):
    y = x.right
    x.right = y.left
    y.left = x
    x.h = max(height(x.left), height(x.right)) + 1
    y.h = max(height(y.left), height(y.right)) + 1
    return y  # y is the new root
```

一共可能有四种情况：

1. LL：x.left.h>x.right.h 且 x.left.left.h>x.left.right.h，则对x进行一次右旋
2. RR：x.right.h>x.left.h 且 x.right.right.h>x.right.left.h，则对x进行一次左旋
3. LR：x.left.h>x.right.h 且 x.left.right.h>x.left.left.h，则对$x.left$进行一次左旋，对x进行一次右旋

```
def left_right_rotate(x):
    x.left = left_rotate(x.left)
    return right_rotate(x)
```

5. RL：x.right.h>x.left.h 且 x.right.right.h>x.right.right.h，则对$x.right$进行一次右旋，对x进行一次左旋

```
def right_left_rotate(x):
    x.right = right_rotate(x.right)
    return left_rotate(x)
```

完整的Balance(x)

```
# 节点类定义
class Node:
    def __init__(self, key):
        self.key = key  # 节点的值
        self.left = None  # 左子树
        self.right = None  # 右子树
        self.h = 1  # 节点的高度

# 获取节点高度的辅助函数
def height(node):
    if node is None:
        return 0
    return node.h

# 计算节点的平衡因子
def get_balance(node):
    if node is None:
        return 0
    return height(node.left) - height(node.right)

# 右旋操作（LL情况）
def right_rotate(y):
```

```
    x = y.left
    T2 = x.right

    # 执行旋转
    x.right = y
    y.left = T2

    # 更新高度
    y.h = max(height(y.left), height(y.right)) + 1
    x.h = max(height(x.left), height(x.right)) + 1

    # 返回新的根节点
    return x

# 左旋操作（RR情况）
def left_rotate(x):
    y = x.right
    T2 = y.left

    # 执行旋转
    y.left = x
    x.right = T2

    # 更新高度
    x.h = max(height(x.left), height(x.right)) + 1
    y.h = max(height(y.left), height(y.right)) + 1

    # 返回新的根节点
    return y

# 平衡函数
def BALANCE(x):
    # 更新当前节点的高度
    x.h = max(height(x.left), height(x.right)) + 1

    # 计算平衡因子
    balance = get_balance(x)

    # 如果平衡因子大于 1，说明左子树高度大于右子树
    if balance > 1:
        # 检查是否为左左情况
        if get_balance(x.left) >= 0:
            return right_rotate(x)  # 右旋
```

```
        # 左右情况，先对左子树进行左旋，再右旋
        else:
            x.left = left_rotate(x.left)
            return right_rotate(x)

    # 如果平衡因子小于 -1，说明右子树高度大于左子树
    if balance < -1:
        # 检查是否为右右情况
        if get_balance(x.right) <= 0:
            return left_rotate(x)  # 左旋
        # 右左情况，先对右子树进行右旋，再左旋
        else:
            x.right = right_rotate(x.right)
            return left_rotate(x)

    # 如果平衡因子在 [-1, 1] 之间，不需要旋转，返回节点本身
    return x
```

(c):

1. 思路：先以二叉搜索树的规则插入节点，再通过左旋右旋保持平衡
2. AVLINSERT(x, z)的伪代码实现

```
def AVLINSERT(x, z):
    if x is None:
        return z  # 新节点成为根
    elif z.key < x.key:
        x.left = AVLINSERT(x.left, z)  # 插入左子树
    else:
        x.right = AVLINSERT(x.right, z)  # 插入右子树

    x.h = max(height(x.left), height(x.right)) + 1  # 更新高度
    return BALANCE(x)  # 平衡子树
```

(d):

显然每次插入只会出现：LL、RR、LR、RL。四种情况之一，每种情况只需要至多两次rotation即可使AVL恢复平衡，而插入节点时遵循二叉搜索树的规则，(a)已证明$h = O(\log n)$，则显然时间复杂度为$O(\log n)$且旋转次数为$O(1)$。

Problem 6

(a):

1. RBINSERT：插入节点初始设置为红色，不会改变T.bh，在插入后检测红黑树的性质，RBINSERT会通过旋转和重涂维护T.bh。
    1. 当插入节点的父节点为黑时直接插入即可，T.bh不会改变
    2. 当插入节点的叔叔节点为红色时，会让叔父爷三个节点变色，再设置爷爷节点为插入节点，进行判定，这也能维护当前分枝内的T.bh一致
    3. 当插入节点的叔叔节点为黑色时，会进行旋转(LL RR LR RL)，接着变色，这也能维持T.bh
    综上，三种情况都未开辟额外空间且成功维护了T.bh，可以证明RBINSERT满足这个性质
2. RBDELETE：删除过程较为复杂，但是原理和RBINSERT类似，都会在删除后进行对红黑树的维护，保证root到每个叶子(BLACK)节点的路径长度一致，且不会开辟额外空间。
3. 计算每个节点的黑高度只需要利用公式：$bh(v) = 1 + max(bh(v.left), bh(v.right))$，因此可以在$O(1)$，时间内完成访问与更新。

(b):

1. 思路：只需要沿着根节点向右查找，设置初始高度为0，若右节点为黑，则高度加一，直至高度为$T_2.bh$，此时返回当前的高度即可。
2. 伪代码

```
def find_T2_hight_node(node,T_2.bh):
        target = T_2.bh
        current_hight = T_1.bh
        current = node
        while current_hight != target:
                current = current.right
                if current.color == black:
                        current_hight --
        return current.value
```

(c):

1. 思路：将x设置为根节点，并将x的左子树的根设置为y，右子树的根设置为T_2的根即可。
2. 伪代码

```
def replace_Ty(x,T_y_root):
        y_c = copy.deepcopy(y)
        x.left = y_c
        x.right = T_y_root
        y = x
```

(d):

结论：将x设置为红色

1. 性质1:节点为红色或者黑色，显然满足
2. 性质3: 所有叶子都是黑色(叶子是NIL节点)，x节点有左右子树而左右子树都满足这个性质，因此x设置为红色后，整个树叶满足此性质
3. 性质5:从任一节点到其每个叶子的所有简单路径都包含相同数目的黑色节点。原本的T_y变为了x的左子树，T_2变为了右子树，因为replace_Ty的设置，T_y和T_2的黑高相同，且都符合原本树的性质5，显然因此设置x为红后不会路径中的黑节点的数量，因此不会改变黑高，即不会改变性质5。

---

1. 性质2:根是黑色。显然
2. 性质4:每个红色节点必须有两个黑色的子节点。设置x为插入点，向上重新着色，因为红黑树的高度至多为$\log n$，因此着色的时间复杂度为$O(\log n)$，插入T_2后树仍需要至多两次旋转即可，因此维护性质2和4时间复杂度至多为$O(\log n)$。

(e):

若假设改为$T_1.bh \leq T_2.bh$，则将之前所有操作中的T_1和T_2互换即可。

(f):

(b)中find_T2_hight_node的时间复杂度为$O(\log n)$，(c)中将T_2嫁接到新的x上时间复杂度为$O(1)$，(d)中维护红黑树的性质时间复杂度为$O(\log n)$，因此RBJOIN就是这几个操作的综合时间复杂度显然为$O(\log n)$.

Problem 7

(a):

伪代码：

```
import random

# 定义Treap的节点
class TreapNode:
    def __init__(self, key, priority):
        self.key = key
        self.priority = priority
        self.left = None
        self.right = None
```

```
def RANDOM():
    # 假设 RANDOM() 可以在 O(1) 时间内生成一个随机数

# 右旋操作
def right_rotate(y):
    x = y.left
    T2 = x.right
    # 进行旋转
    x.right = y
    y.left = T2
    return x

# 左旋操作
def left_rotate(x):
    y = x.right
    T2 = y.left
    # 进行旋转
    y.left = x
    x.right = T2
    return y

# 插入节点并保持堆和BST性质
def insert_treap(root, key):
    if root is None:
        # 为新节点生成随机优先级
        priority = RANDOM()
        return TreapNode(key, priority)

    # 根据二叉搜索树的性质插入新节点
    if key < root.key:
        root.left = insert_treap(root.left, key)
        # 检查是否需要右旋
        if root.left is not None and root.left.priority > root.priority:
            root = right_rotate(root)
    else:
        root.right = insert_treap(root.right, key)
        # 检查是否需要左旋
        if root.right is not None and root.right.priority > root.priority:
            root = left_rotate(root)

    return root

# 构建随机Treap
```

```
def construct_random_treap(A):
    root = None
    for key in A:
        root = insert_treap(root, key)
    return root
```

(b):

1. 很大概率下最大层数为 O(logn)：
   在skip_list中，每一层的节点数量是上一层节点数量的二分之一。因此，第 i 层的节点数量期望为 $\frac{n}{2^i}$。而第一层大概率只有两个节点，因此有等式$\frac{n}{2^i}=2$，则可得出$i=\log n/2$，则层数为$O(\log n)$。
2. skip_list的期望节点总数为O(n):
   已知每个节点出现在上一层的概率为$\frac{1}{2}$，则进入第i层第概率为$\frac{1}{2^i}$，因此第i层节点的期望数量为$\frac{n}{2^i}$，对i从1到$\lfloor \log n \rfloor$求和，即可求得总节点数为2n，即为$O(\log n)$。