第三次实验报告

胡延伸 PB22050983

实验内容

编码实现红黑树的插入算法，使得插入后依旧保持红黑性质。

实验目的

- 程序输入一个文件，包含各个结点的 key 值;
- print输出：RB_INSERT_FIXUP算法历经的情况种类数
- 文件输出：将插入完成后的红黑树进行“先序遍历（NLR）”，“中序遍历（LNR）”和“层次遍历（Level-Order Traverse）”，并将相应的遍历序列输出到文件中。

算法设计思路

首先将红黑树 `T` 当成一棵普通的二叉搜索树，然后将待插入的结点 `z` 按照普通二叉搜索树的插入方法插入后，将其着为红色。为保证红黑性质能继续保持，最后再调用辅助程序 `RB_INSERT_FIXUP` 进行必要的旋转与重新着色。

实验源码

红黑树结构定义

```
# 定义结点
class Node:
    def __init__(self, key: int, right, left, parent, color: str):
        self.key = key
        self.right = right
        self.left = left
        self.parent = parent
        self.color = color

class REDBLACKTREE():
    # 初始化红黑树
    def __init__(self):
        self.nil = Node(None, None, None, None, 'black')
```

```
        self.root = self.nil

    # 定义 RB_INSERT 方法插入结点
    def RB_INSERT(self, z):
        y = self.nil
        x = self.root
        while x != self.nil:
            y = x
            if z.key < x.key:
                x = x.left
            else:
                x = x.right
        z.parent = y
        if y == self.nil:
            self.root = z
        elif z.key < y.key:
            y.left = z
        else:
            y.right = z
        z.left = self.nil
        z.right = self.nil
        z.color = 'RED'
        # 在完成类似于二叉搜索树的插入方法后，进行必要的旋转与着色
        self.RB_INSERT_FIXUP(z)

    def RB_INSERT_FIXUP(self, z):
        while z.parent.color == 'RED':
            if z.parent == z.parent.parent.left:
                y = z.parent.parent.right
                if y.color == 'RED':                    # case 1
                    print('Case 1')
                    z.parent.color = 'BLACK'
                    y.color = 'BLACK'
                    z.parent.parent.color = 'RED'
                    z = z.parent.parent
                else:
                    if z == z.parent.right:             # case 2，将其转换为
case 3
                        print('Case 2')
                        z = z.parent
                        self.LEFT_ROTATE(z)
                    print('Case 3')
```

```
                    z.parent.color = 'BLACK'            # case 3
                    z.parent.parent.color = 'RED'
                    self.RIGHT_ROTATE(z.parent.parent)
            else:
                y = z.parent.parent.left                # 与上述情况完全对称
                if y.color == 'RED':                    # case 4
                    print('Case 4')
                    z.parent.color = 'BLACK'
                    y.color = 'BLACK'
                    z.parent.parent.color = 'RED'
                    z = z.parent.parent
                else:
                    if z == z.parent.left:              # case 5，将其转换为
case 6
                        print('Case 5')
                        z = z.parent
                        self.RIGHT_ROTATE(z)
                    print('Case 6')
                    z.parent.color = 'BLACK'            # case 6
                    z.parent.parent.color = 'RED'
                    self.LEFT_ROTATE(z.parent.parent)
        self.root.color = 'BLACK'

        # 上述左旋转，与右旋转的定义如下：
        def LEFT_ROTATE(self, x):
        y = x.right
        x.right = y.left
        if y.left != self.nil:
            y.left.parent = x
        y.parent = x.parent
        if x.parent == self.nil:
            self.root = y
        elif x == x.parent.left:
            x.parent.left = y
        else:
            x.parent.right = y
        y.left = x
        x.parent = y

    def RIGHT_ROTATE(self, y):
        x = y.left
        y.left = x.right
```

```
        if x.right != self.nil:
            x.right.parent = y
        x.parent = y.parent
        if y.parent == self.nil:
            self.root = x
        elif y == y.parent.right:
            y.parent.right = x
        else:
            y.parent.left = x
        x.right = y
        y.parent = x
```

先序遍历与中序遍历

这两者均为简单的递归函数:

```
# 先序遍历
def NLR(self, node):
    self.write_lines.append((str(node.key) if node!= self.nil else '#') + ',
' + node.color+'\n')
    if node.left != self.nil:
        self.NLR(node.left)
    if node.right != self.nil:
        self.NLR(node.right)

# 中序遍历
def LNR(self, node):
    if node.left != self.nil:
        self.LNR(node.left)
    self.write_lines.append((str(node.key) if node!= self.nil else '#') + ',
' + node.color + '\n')
    if node.right != self.nil:
        self.LNR(node.right)
```

层次遍历

层次遍历需要引入辅助的队列结构，每次取出一个元素，输出它的 key 值和颜色，然后将其左右孩子入队，直到队列为空:

```
    # 定义层次遍历输出函数
    def level_order_traverse(self, file='LOT.txt'):  # 层次遍历的结果保存在
LOT.txt 文件中
        # 先清空原有的 lines
        self.write_lines = []
        node = self.root
        NodeQueue = []
        NodeQueue.append(node)
        while NodeQueue:
            # 先推出队列第一个元素
            node = NodeQueue.pop(0)
            # 再将其左右孩子入队
            if node.left≠ self.nil:
                NodeQueue.append(node.left)
            if node.right≠ self.nil:
                NodeQueue.append(node.right)
            self.write_lines.append((str(node.key) if node≠ self.nil else
'#') + ', ' + node.color+'\n')
        with open(file, 'w') as file:
            file.write(''.join(self.write_lines))
        file.close()
```

算法正确性测试

中序遍历结果完全正确，key 按照递增的顺序排列，见输出文件,如下图参考:

```
0, BLACK
1, RED
2, BLACK
3, RED
4, BLACK
5, BLACK
6, RED
7, BLACK
8, RED
9, BLACK
10, RED
11, BLACK
12, RED
13, BLACK
14, BLACK
15, RED
16, BLACK
17, RED
18, RED
19, BLACK
```

层次遍历和先序遍历的验证调用库函数 RBTree, 经过比对，输出结果一致，表明算法实现正确。库函数的源码见文件 `test.py` .

遇到的困难及收获

本实验遇到的困难有：

- 考虑插入结点时对称的情况容易误认结点的属性及位置。
- 写入文件时发生错乱，不应该多次打开文件添加内容，而应该先写好再一次性添加进文件中。

收获有：

- 理解红黑树的性质，深入了解了红黑树插入的算法。
- 对递归函数的应用有了进一步的认知。