数据结构与算法 - 综合训练 1
用树解决问题

吴天阳 2204210460 强基数学 002

1 实验目的

实验 1：实现二叉检索树 (Binary Search Tree, BST)，需要包含以下 8 种结构

```python
class BSTBase():  # BST 基类
    def insert(self, key, value): pass  # 插入 (key,value) 键值对
    def remove(self, key): pass  # 删除 key 节点，若找到并删除节点，则返回对应的 value，若无该节点则返回 None
    def search(self, key): pass  # 查询 key 对应的 value，若无 key 节点，则返回 None
    def update(self, key, value): pass  # 更新 key 对应的 value，若无 key 节点，返回 False，否则返回 True
    def isEmpty(self): pass  # 判断二叉搜索树是否为空
    def clear(self): pass  # 重新初始化 BST
    def showStructure(self, file): pass  # 输出当前二叉树的节点总数和高度到文件 file 中
    def printInorder(self, file): pass  # 输出二叉树的中序遍历到文件 file 中
```

读入命令文件 `BST_testcases.txt` 并处理执行，将输出运行结果到文件中，并与 `BST_result.txt` 进行比较，判断程序是否正确.

实验 2：使用 BST 为文稿建立单词索引表，基于给出的英文文本 `article.txt`，记录每个单词在文本中出现的行号（编号从 1 开始）.

2 实验原理

二叉搜索树是一种二叉树形式的数据结构，支持**插入**、**查找**、**删除**键值对的功能，对于一个有 n 个节点的二叉搜索树，每次操作的最优复杂度为 $\mathcal{O}(\log n)$，最坏为 $\mathcal{O}(n)$，随机构成一颗二叉树的期望高度为 $\mathcal{O}(\log n)$.

注：平衡树是在二叉搜索树基础上进行的改进，可以保证每次查找的复杂度为 $\mathcal{O}(\log n)$.

二叉搜索树定义如下：

1. 空树是二叉搜索树.
2. 若二叉搜索树的左子树非空，则左子树上所有点的 key 值均小于根节点的 key 值.
3. 若二叉搜索树的右子树非空，则右子树上所有点的 key 值均大于根节点的 key 值.
4. 二叉搜索树的左右子树均为二叉搜索树.

实现以上操作基本均通过递归即可实现，节点之间的关联可用指针实现，具体实现请见下文.

3 实验步骤与结果分析

使用 Python 3.9.12进行实现. 在 Python 中数值类型数据int, float, str无法直接传入实参，但是使用 class, list, dict则默认传入实参，所以需要利用该性质实现指针操作.

代码文件总共有三个：my_bst.py 包含 BST 核心类，main1.py, main2.py 分别为实验 1 和实验 2 的代码，对应的输出文件分别为 my_result1.txt, my_result2.txt.

3.1 实现 BST 类

3.1.1 初始化

首先创建节点类 Node，具体有以下属性：

- key(int)：存储 key 值.
- val0：用于初始化 val 值.（用于实验 2，可以为 list，存储多个 value 值）
- val：存储 val 值.
- child(list)：长度为 2 的 list，child[0],child[1] 分别表示左右子节点.

```
class Node():  # 节点子类
    val0 = 0
    def __init__(self):
        self.key = None  # 初始化键值
        self.child = [None, None]  # 初始化左右孩子节点
        if isinstance(self.val0, list):
            self.val = []  # 由于 list 按照实参赋值，必须重新创建空 list
        else: self.val = self.val0
```

初始化 BST 类

```
def __init__(self, val0=0):
    self.Node.val0 = val0  # val0 为每个节点值的初值
    self.file = None  # 将要写入的文件
    self.root = None  # 创建根节点
    self.height = 0  # 树的高度
    self.size = 0  # 树的节点总数
```

判断二叉搜索树是否为空，只需判断 BST.root 是否为None即可.

```
def isEmpty(self):
    return self.root == None
```

重新初始化整个 BST，由于需要删除掉全部节点，所以需要递归删除，这里使用中序遍历中的递归顺便完成该任务：

```
def dfs(self, p, delete_node=False):  # 输出中序遍历结果
    if p.child[0]:
        self.dfs(p.child[0], delete_node)
    if not delete_node:
        self.file.write('[{} ---- < {} >]\n'.format(p.key,
        ↪  str(p.val)[1:-1]))  # 在文件中直接写入
```

```
    if p.child[1]:
        self.dfs(p.child[1], delete_node)
    if delete_node:  # 用于清空整棵树
        del p

def clear(self):
    self.dfs(self.root, delete_node=True)
    self.__init__()  # 调用初始化函数, 清空 BST

def printInorder(self, file):
    if file is None:
        return None
    self.file = file
    self.dfs(self.root)
    return   # 返回中序遍历
```

3.1.2 加入节点

直接通过递归找到对应键值位置，然后修改 value 值即可.

```
def add(self, p, key, val):
    if p is None:  # 当前节点为空节点, 开始创建
        p = self.Node()
        p.key = key
    if key == p.key:  # 找到 key 值对应节点, 更新节点 val
        if isinstance(p.val, list):  # 如果 val 当前是列表, 则加入值
            p.val.append(val)
        else:  # 否则直接修改当前值
            p.val = val
    elif key < p.key:  # key 节点在左子树中
        p.child[0] = self.add(p.child[0], key, val)
    else:  # key 节点在右子树中
        p.child[1] = self.add(p.child[1], key, val)
    return p

def insert(self, key, val):
    if key is None or val is None:
        return
    self.root = self.add(self.root, key, val)  # 每次从根节点开始查找插入位置
```

3.1.3 查询节点

直接通过递归搜索即可.

```
def find(self, p, key):
    if p == None:  # 空节点
        return None
    elif key == p.key:  # 找到了 key 节点
        return p.val
    elif key < p.key:  # key 节点在左子树中
```

```
        return self.find(p.child[0], key)
    return self.find(p.child[1], key)  # 否则只能在右子树中

def search(self, key):
    if key is None:
        return None
    return self.find(self.root, key)
```

3.1.4 删除节点

删除节点稍微有点麻烦，若删除的节点有两个子节点，为了保持二叉搜索树的性质，需要用左子树中的最大值，或者右子树的最小值替代删除掉的节点；若删除的节点仅有一个儿子，则直接用它代替即可；若删除的节点为叶子结点，直接删去即可.

```
def delete_min(self, p):  # 查找最小值
    if p.child[0] is None:  # 找到最小值
        return p, p.child[1]  # 返回的第一个参数为找到的最小值点，第二个参数为更新点编号
    min_p, update_p = self.delete_min(p.child[0])
    p.child[0] = update_p  # 用第二个参数更新该点
    return min_p, p  # 返回的第一个参数为找到的最小值点，第二个参数为更新点编号

def delete(self, p, key):
    if p.key == key:
        if p.child[0] and p.child[1]:  # 如果有两个儿子节点，就需要找到左子树中的最大值或右子树的最小值替代
            # 这里找右子树最小值替代
            min_p, update_p = self.delete_min(p.child[1])
            min_p.child[0] = p.child[0]
            min_p.child[1] = update_p
            del p  # 删除掉当前节点 p
            return min_p
        else:
            ret = p.child[0] if p.child[0] else p.child[1]
            del p  # 删除掉当前节点 p
            return ret
    elif key < p.key:  # key 节点在左子树中
        p.child[0] = self.delete(p.child[0], key)
    else:  # key 节点在右子树中
        p.child[1] = self.delete(p.child[1], key)
    return p

def remove(self, key):
    val = self.search(key)
    if key is None or val is None:  # 如果找不到该 key 值也返回 False
        return None
    self.root = self.delete(self.root, key)
    return val
```

3.1.5 查询 BST 结构

直接通过递归即可完成.

```
def struct(self, p, h):  # 查找 bst 的节点数和高度
    self.size += 1  # 总节点数 +1
    self.height = max(self.height, h)  # 更新树的深度
    if p.child[0]:
        self.struct(p.child[0], h + 1)
    if p.child[1]:
        self.struct(p.child[1], h + 1)

def showStructure(self, file):  # 返回树的总节点数, 返回树的高度
    if file is None:
        return None
    self.size, self.height = 0, 0  # 先初始化为 0
    self.struct(self.root, 1)
    file.write('----------------------------\n')
    file.write(f'There are {self.size} nodes in this BST.\nThe height of
    ↪ this BST is {self.height}.\n')
    file.write('----------------------------\n')
```

3.2 实验 1

主要是对读入字符串的划分和条件判断，执行对应的 BST 函数即可.

```
import my_bst

with open('BST_testcases.txt', 'r', encoding='utf-8') as file,\
        open('my_result1.txt', 'w', encoding='utf-8') as outfile:
    bst = my_bst.BST()
    while True:
        line = file.readline()
        if not line:
            break
        opt = line[0]
        if opt == '#':
            bst.showStructure(outfile)
            continue
        key = line.split(' ')[1]  # 提取 key 值
        if opt == '+':
            value = line.split('\"')[1]  # 提取 value 值
            bst.insert(key, value)
        elif opt == '-':
            value = bst.remove(key)
            if value is not None:
                outfile.write(f'remove success ---{key} {value}\n')
            else:
                outfile.write(f'remove unsuccess ---{key}\n')
        elif opt == '?':
            value = bst.search(key)
```

```
            if value is not None:
                outfile.write(f'search success ---{key} {value}\n')
            else:
                outfile.write(f'search unsuccess ---{key}\n')
        elif opt == '=':
            value = line.split('\"')[1]  # 提取 value 值
            flag = bst.update(key, value)
            if flag:
                outfile.write(f'update success ---{key} {value}\n')
            else:
                outfile.write(f'update unsuccess ---{key}\n')
```

3.3 实验 2

主要先对文本进行清洗，清洗方法非常简单，按空格划分出单词，然后去除掉首尾的非字母符号即可.

```
import my_bst

def wash(word):  # 将其他字符都去掉, 只剩下拉丁字母
    while len(word):
        if not word[-1].isalpha():
            word = word[:-1]
        elif not word[0].isalpha():
            word = word[1:]
        else:
            break
    return word

with open('article.txt', 'r', encoding='utf-8', errors='ignore') as file:
    bst = my_bst.BST(val0=[])
    cnt = 0  # 用于记录行数
    while True:
        # print(cnt)
        line = file.readline()
        if not line:
            break
        cnt += 1
        line = wash(line)  # 将每一行也洗一下, 把多余的空格去掉
        words = line.split(' ')  # 将一行中的单词分离出来
        for key in words:  # 逐个遍历
            key = wash(key)  # 将 key 中的其他标点去掉, 只剩下单词
            if not len(key):  # 洗没了
                continue
            bst.insert(key, cnt)

with open("my_result2.txt", 'w') as file:
    bst.printInorder(file)
```

4 结论与讨论

最后通过比对，my_reslut1.txt 与 BST_result.txt 完全一致，说明 BST 算法应该基本正确，实验 2 的输出文件为 my_result2.txt，去重后总计有 16580 个单词.

本次实验，我学会了如何使用 Python 实现 BST 的指针操作，相比 C 语言虽然不直观，但是实现效果非常简洁. Python 在处理文本上相较于 C 也更为方便.