CS202: Data Structure and Algorithm Analysis

update on 2022/1/8 10:22

2022 Spring, Complied by Hongfei Yan

以下内容，主要摘录自“数据结构与算法（B），谢正茂，2021年春季”.

目标：

写程序处理数据、分析数据的能力，能把“大数据”工具用在各自的领域。
进一步学习人工智能/机器学习，很好的起点。

内容：

算法基本概念，算法复杂度

线性表

递归与动态规划

KMP算法

排序与查找

树及算法

图及算法

课件下载地址：

https://disk.pku.edu.cn:443/link/05C784FCC089D4B527278B041CE480E2

Valid Until: 2023-02-28 23:59

Password: K4ns

教材：

Problem Solving with Algorithms and Data Structures Using Python [1]

Introduction to Algorithms [2]

课程成绩：

- 满分 100 分

  期末理论笔试:40%，平时课堂 + 作业:30%，上机编程:30%
- 附加分 3 分 微信答疑 3 分

```
关于本学期上机考试的安排，发布时间：2021年5月24日
各位同学，
    拟于2021年6月2日（周三）下午5-6节（下午1点-3点）进行本学期《算法与数据结构B》上机考试，请大家前往平时上机的机房（计算中心7号机房）进行考试。现将考试主要注意事项通知如下：

1、请按照本学期发布的座位号就坐，就坐后提前检查自己的机器编程环境，如有问题，请向监考老师说明，同意后才可更换座位；
2、考试网站为http://cesedsa.openjudge.cn，请大家熟记自己的用户名和密码，且在考场上不要尝试换用户名登录，否则计算机将被锁，无法继续考试；
3、考试时间为2小时；请将学生证放到桌面上，并自觉遵守考试纪律；
4、本次考试题目共6题，按难易程度分为3个等级，请优先考虑较容易的题目；
5、考试可以带纸质书籍、笔记、个人总结等，但不许相互借用。考试时间有限，建议不要带过多的资料；
6、不准使用任何自带电子设备（手机、优盘或其它移动存储设备）。一旦发现作弊，0分论处；
7、遇到其他问题，请及时与监考老师取得联系。

为了帮助大家熟悉考试环境和要求，我们将在5月26日的同一时间准备一次模拟考试，请同学们尽量参加。
```

```
关于参加本学期机考统计的通知，发布时间：2021年5月24日
同学们：
    为了准备很快就要进行的期末机考，现在进行参考身份统计。请同学们确保已经完成以下事项：
    1、注册并加入openjudge网站：http://cesedsa.openjudge.cn/,
    2、把自己的昵称修改为“学号_姓名”的格式，比如“[2000011507_黄一凡](http://openjudge.cn/user/1072906/in/group-480/)”
    3、加入班级“数算B-21春季2班”
    谢谢大家的配合！
```

```
第七章《树及其算法》课件更新，发布时间：2021年5月25日
增加了“二叉树与树林/树的转换”内容
```

关于本课程上机时间与其他专业/必修课冲突的问题，发布时间：2021年3月6日

同学们好！

近来有同学发信问我“课程上机时间与其他专业/必修课冲突的问题“的问题，现在这里统一答复：

1、学习数算B，除了上正课之外，同学们的动手编程能力也需要练习提高。编程方面不同背景的同学基础相差很大，上机课主要是用来给基础差的同学答疑。发现一些较普遍的问题，也会拿到正课上讲。因此基础好的同学可以自己安排其他时间练习编程，完成作业，只是偶尔来一下也是没有问题的。但编程能力是课程的考核项目之一，这些同学需要保证最后不会“因为不能来答疑，编程方面仍有困难，影响了期末成绩“。

2、期末上机考试的时间，很可能和上机课的时间一样。这个没有其他办法：上机考试必须按时来，不来没成绩。

我内心非常希望能够和你们共同学习一个学期。但还是请同学们认真思考后，根据自己的实际情况做出决定。

共八次作业

作业1 问题抽象与 Git 使用

发布时间：March 15, 2021，截止时间:3月21日24 时

- 十字路口的信号灯问题 2pts

有 A,B,C,D 四条道路相交的十字路口是最常见的路口，并且满足下 面的条件:

A,B,C,D 分别通往东、南、西、北四个方向

所有道路都是双向通行，没有单行道

车辆靠道路右侧通行

请仿照课件中的五叉路口例子，画出十字路口行驶冲突的模型，并给出满足安全性的信号灯分组最优解。

- 由认识关系组成的班级社交网络(调查) 1pt

通过认识关系，班上的所有同学组成了一张有向图。本学期我们将利用图的算法来研究这张图，请同学们按下面的格式提供认识关系:

第一行:我自己的名字

第二行:我认识的班上三个同学的名字(尽量提供，如果没有这里就 写上 “没有”)。

- 作业提交方式：

通过 Git 方式提交到服务器

建立子目录 hw1，目录下有如下文件:

行程冲突的模型图: crossroad_model.png

信号灯的最优分组方案: best_group.txt

我认识的三位同班同学: iknows.txt

作业2 Python 语言基础掌握

发布时间：March 18, 2021，作业截止时间:3 月 23 日 24 时

评分标准

合格	
猜数字格式化语法	分数:0.5 (16.66666%)是否将猜数字游戏的代码中的print函数中的字符串进行格式化处理
字符计数格式化语法	分数:0.5 (16.66666%)是否将字符计数的代码中的print函数中的字符串进行格式化处理
程序正确运行	分数:2 (66.66666%)提交的程序是否能正确运行，每个程序0.5分，共计2分

- 猜数游戏 1pt

输入下列程序，保存为 hw2/guess.py，并运行通过:

```python
# 猜数字游戏
import random

secret = random.randint(1, 100)
print('''猜数游戏!
我想了一个1-100的整数，你最多可以猜6次，
看看能猜出来吗? ''')
tries = 1
while tries <= 6:
    guess = int(input("1-100的整数，第%d次猜，请输入: " % (tries,)))
    if guess == secret:
        print("恭喜答对了！你只猜了%d次！\n就是这个: %d! " % (tries, secret))
        break
    elif guess > secret:
        print("不好意思，你的数大了一点儿！")
    else:
        print("不好意思，你的数小了一点儿！")
    tries += 1
else:
    print("哎呀！怎么也没猜中！再见！")
```

- 归并排序 1pt

输入下列程序，保存为 hw2/merge_sort.py，并运行通过:

```python
# merge sort
# 归并排序
import random


def merge_sort(data_list):
    if len(data_list) <= 1:
        return data_list
    middle = int(len(data_list) / 2)
    left = merge_sort(data_list[:middle])
    right = merge_sort(data_list[middle:])
    merged = []
    while left and right:
        merged.append(left.pop(0) if left[0] <= right[0] else right.pop(0))
    merged.extend(right if right else left)
    return merged


data_list = [random.randint(1, 100) for _ in range(50)]
print(merge_sort(data_list))
```

- 一年的第几天?1pt

输入下列程序，保存为 hw2/date.py，并运行通过:

```python
# 输入某年某月某日，判断这一天是这一年的第几天?
import datetime

dtstr = input('Enter the datetime:(20170228):')
dt = datetime.datetime.strptime(dtstr, "%Y%m%d")
another_dtstr = dtstr[:4] + '0101'
another_dt = datetime.datetime.strptime(another_dtstr, "%Y%m%d")
print(int((dt - another_dt).days) + 1)
```

- 字符计数 1pt

输入下列程序，保存为 hw2/count.py，并运行通过:

```
#输入一行字符，分别统计出其中英文字母、空格、数字和其它字符的个数。
import string

s = input('input a string:')
letter = 0
space = 0
digit = 0
other = 0
for c in s:
    if c.isalpha():
        letter+=1
    elif c.isspace():
        space+=1
    elif c.isdigit():
        digit+=1
    else:
        other+=1
print('There are %d letters,%d spaces,%d digits \
and %d other characters in your string.'\
      %(letter,space,digit,other))
```

- 特别要求

把上面四个程序中，原来的字符串格式化语法

”I have %d books and %d pens” % (3,4)

替换成

”I have {0} books and {1} pens”.format(3,4)

作业3 算法计量

March 24, 2021，截止时间:3 月 28 日 24 点

评分标准

	合格
接收用户输入	分数:0.5 (16.66666%)接收用户输入的 minN, maxN, step。
构造列表	分数:0.5 (16.66666%)常见错误：将列表构造为 list(range(minN, maxN, step))，因为列表的大小应该为 N。
重复实验	分数:0.5 (16.66666%)设置 Timer 的运行次数或自行计时的运行次数使得得到的结果较为平滑。
结果	分数:0.5 (16.66666%)合理使用 Timer 和随机数，使结果正确，即用下标访问（indexing）为 O(1)、搜索元素（searching）为 O(N)。 常见错误： 1. 使用 Timer('indexing({0}, {1})'.format(random.randint(0,N-1), mylist),'from **main** import indexing')，字符串 format 一个长列表的操作是 O(N) 的，「污染」了用下标访问的 O(1) 操作，使得得到的结果不正确； 2. Timer 每次执行搜索或用下标访问时所用的随机数应该是不同的，否则得到的结果波动太大。
绘图	分数:1 (33.33333%)用 matplotlib.pyplot 或其他作图工具正确展示结果。

- 比较 List 两种操作的算法复杂度 3pts

大小为 N 的整数列表 mylist=[...]，两种操作分别是:

按索引取值:mylist[idx], idx 为列表的下标

检索:mylist.index(item)，item 为列表的元素

计量两种操作的运行时间，在一张图中画出它们随 N 变化的曲线

设计实验需要考虑的因素:

每个操作需要重复一定次数，以避免随机误差
idx, item 需要随机产生，避免 cache 对计量的影响
N 在 min, max 之间以步长 step 进行变化，这三个数由用户输入

作业提交形式，Git:hw3/list2op.py

作业4 比特币挖矿 & 栈的应用

April 2, 2021，截止时间:2021 年 4 月 11 日 24 点(逾期扣分/天:0.1 => 0.5)

评分标准

	合格
比特币：尝试回答问题	分数:1.5 (15.00%)见字给分，复杂度0.5，np/p和关系 0.5，时间估计0.5
比特币：答案正确	分数:1.5 (15.00%)复杂度0.5，np/p和关系 0.5，时间估计0.5
中缀转前缀：输入输出	分数:0.5 (5.00%)需严格符合规定
中缀转前缀：代码质量	分数:1.5 (15.00%)
中缀转前缀：测试样例	分数:2 (20.00%)0.2 * 10
HTML：输入输出	分数:0.5 (5.00%)需严格符合规定
HTML：代码质量	分数:0.5 (5.00%)
HTML：测试样例	分数:2 (20.00%)0.2 * 10

- 比特币挖矿的算法复杂度 3pts

根据卡矿的示例代码，分析说明下面问题:
1: validate 和 find_nonce 的复杂度分别是多少?它们是什么关系?分别属于 NP 还是 P?
2: 如果想把“Alice send .1 coins to Kate”改成“Alice send 100 coins to Kate”，计算一个新 nonce 来保持“块哈希”值不变，算法 复杂度是多少?
请把问题的答案写到名为 btc.txt 的文件中

- 栈的应用-中缀转前缀 4pts

实现一个“中缀转前缀”算法
可能需要从右往左扫描
需要多次全量反转顺序
实现算法的代码文件为 in2pre.py

输入格式:从标准输入读取一行中缀表达式，格式如课件所述，字符 之间由空格分隔，操作符包括 +-*/和小括号，操作数为单个字母。

输出格式:输出一行，为转换后的前缀表达式，字符间由空格分隔， 其中操作数顺序与必须与输入中缀表达式的顺序相同。(答案唯一， 不许使用交换律)

样例输入:(A + B) * (C + D)

样例输出:* + A B + C D

- 栈的应用-html 文档标记检查 3pts

扩展括号匹配算法，用来检查 HTML 文档的标记是否匹配。

HTML 标记应该成对、嵌套出现

开标记是 <tag> 这种形式，闭标记是 </tag> 这种形式
读入一个 HTML 文件，算法检查是 否有标记不匹配的情况

代码文件为 html.py

输入格式:从标准输入读取 html 格 式文本，其中每组成对的标记名称均 为字母/数字组合，可能包含多行， 可能包含多余的空格和空行

输出格式:若输入的 html 括号匹配 是合法的，输出 yes，否则输出 no

```
<html>
   <head>
      <title>
         Example
      </title>
   </head>
   <body>
      <h1>Hello, world</h1>
   </body>
</html>
```

刷题网站:https://leetcode-cn.com/problemset/all/

链表题	栈与队列
2 两数相加	20 有效的括号
19 删除链表的倒数第 N 个结 点	150 逆波兰表达式求值
21 合并两个有序链表	155 最小栈
24 两两交换链表中的结点 61 旋转链表	225 用队列实现栈
82 删除链表中的重复元素 206 反转链表	232 用栈实现队列
328 奇偶链表	

作业5 线性表

April 6, 2021，截止时间：4月18日24 时

评分标准

	合格
基数排序正确	分数:2 (20.00%)
为地铁站建立模型	分数:1 (10.00%)
其他考虑因素	分数:0.5 (5.00%)
模拟不同压力下车站的状况	分数:1 (10.00%)
UnorderedList的每一项0.5分	分数:3 (30.00%)
OrderedList实现	分数:0.5 (5.00%)
采用链表来实现Stack和Queue	分数:1 (10.00%)
双链表结构	分数:1 (10.00%)

- (队列)基数排序算法 3pts

实现一个基数排序算法，用于 10 进制的正整数排序。思路是保持 10 个队列(队列 0、队列 1……队列 9、队列 main)，开始，所有的 数都在 main 队列，没有排序。

第一趟将所有的数根据其 10 进制个位(0 ~ 9)，放入相应的队列 0 ~ 9，全放好后，按照 FIFO 的顺序，将每个队列的数合并排到 main 队列
第二趟再从 main 队列队首取数，根据其十位的数值，放入相应队列 0 ~ 9，全放好后，仍然按照 FIFO 的顺序，将每个队列的数合并排 到 main 队列

第三趟放百位，再合并;第四趟放千位，再合并 直到最多的位数放完，合并完，这样 main 队列里就是排好序的数列了

- 队列)西二旗地铁站 3pts

西二旗站是地铁昌平线和 13 号线的交点，附近有大量的科技公司， 拥堵冠绝北京。

```
为地铁站建立模拟模型，除课件中提到的，还有没有其他要考虑的因素？
  编写程序，模拟不同压力下车站的状况，以及需要的应对策略。
```

(附加)思考:昌平线南延，13 号拆成东线和西线，对现状有什么影 响。

- 链表的实现及应用 3pts

实现 UnorderedList 的如下方法: append, index, pop, insert

用于列表字符串表示的 **str** 方法
用于取元素的 **getitem** 方法，类似 python 列表 lst[i]

将 OrderedList 作为 UnorderedList 的子类来实现

采用链表来实现 ADT Stack 和 ADT Queue

目前我们链表采用的是“单链表”结构，其缺点我们也遇到了，就 是无法访问到当前节点的“前驱”节点，请实现“双链表”结构的 UnorderedList

在节点 Node 中增加 prev 变量，引用前一个节点
在 UnorderedList 中增加 tail 变量，引用列表中最后一个节点。

作业6 递归与动态规划

April 19, 2021，截止时间：4月25日 24 时

评分标准

	新手	胜任	精通
绘制树	分数:1 (9.0909%)完成了作业要求的 1 个新特性	分数:2 (18.18181%)完成了作业要求的 2 个新特性	分数:3 (27.27272%)完成了作业要求的 3 个或以上新特性
绘制 Hilbert 曲线	分数:1 (9.0909%)递归算法基本正确，但曲线大小不够或有缺陷	分数:2 (18.18181%)完成了 3 号曲线的绘制	分数:0 (0.00%)
博物馆大盗	分数:2 (18.18181%)动态规划算法基本正确	分数:3 (27.27272%)结果正确（29）	分数:0 (0.00%)
最小编辑距离	分数:2 (18.18181%)动态规划算法基本正确	分数:3 (27.27272%)结果正确（185）	分数:0 (0.00%)

- 递归画图

修改分形树程序，增加如下功 能 3pts

树枝的粗细可以变化，随着 树枝缩短，也相应变细

树枝的颜色可以变化，当树 枝非常短的时候，使之看起 来像树叶的颜色

让树枝倾斜角度在一定范围 内随机变化，如 15 ~ 45 度 之间，左右倾斜也可不一样，

做成你认为最好看的样子

树枝的长短也可以在一定范 围内随机变化，使得整棵树 看起来更加逼真

使用海龟制图，画出如右图所 示希尔伯特曲线 2pts

- 动态规划

动态规划算法:博物馆大盗问题(描述见课件)3pts

给定一个宝物列表 treasure=[{'w':2,'v':3},{'w':3,'v':4},...]

这样 treasure[0]['w'] 就是第一件宝物的重量，等等

给定包裹最多承重 maxw=20

要求写算法输出选取最高总价值的宝物的序号以及价值

动态规划算法:单词最小编辑距离问题(描述见课件)3pts

给定两个单词，要求写算法，得出从源单词变到目标单词所需要的最 小编辑距离，以及编辑操作过程和总得分

作业7 查找与排序

Apr 29, 2021，截止时间: 2021年5月9日 23:59 (逾期扣分:0.5分/天)

bsearch.py, map.py, qsort.py, sort.py四个python文件是实现作业程序的模版。

评分标准

	满分
bsearch-完成情况	分数:2 (16.66666%)
bsearch-测试样例	分数:1 (8.33333%)10 * 0.1
map-完成情况	分数:2 (16.66666%)
map-测试样例	分数:1 (8.33333%)10 * 0.1
set-完成情况	分数:2 (16.66666%)
set-测试样例	分数:1 (8.33333%)10 * 0.1 + x
qsort-完成情况	分数:2 (16.66666%)
qsort-测试样例	分数:1 (8.33333%)10 * 0.1

- 二分查找 3pts

实现二分查找算法，从整数的数组 nums 中，寻找目标 target。如 果 target 在数组中，返回 target 所在的下标;否则返回 −1 − idx， 其中 idx 是 target 应被插入的位置的下标。数组 nums 中不包含重 复元素。

代码模板:bsearch.py，请把代码填写到对应的位置，并在注释中对 自己的实现做出简单的解释。请在提交前做充分的测试以确认程序 的正确性。

```python
# bsearch.py
def bsearch(nums: [int], target: int) -> int:
    return -1

if __name__ == '__main__':
    nums = [int(s) for s in input().split()]
    target = int(input())
    # nums = [1, 3, 5, 6]
    # target = 2
    print(bsearch(nums, target))
```

样例输入:nums = [-1,0,3,5,9,12], target = 9
样例输出:4
样例输入:nums = [1,3,5,6], target = 2
样例输出:-2 (target 应被插入在 3 的位置，下标为 1，-1-1=-2)

可参考 Leetcode 相关题目来测试程序的正确性:

35. 搜索插入位置

704. 二分查找

- 用数据链的冲突解决技术实现 ADT Map 3pts

采用数据链(chaining)的冲突解决技术来实现 ADT Map(功能类 似 python 内置的字典 dict)，其中 key 和 value 均为整数，需要实 现的方法包括:

```
m=Map():创建对象，具体实现为构造函数 __init__(self)
m.put(key, value):将 key-value 关联加入映射 m.get(key):查找关联的数据值，若不存在返回 None
m.remove(key):删除 key 对应的 value 关联，不存在则直接返回
m[key]=value:调用 put 方法。具体实现在 __setitem__(self, key, value) m[key]:调用 get 方法。具体实现在 __getitem__(self, key)
del m[key]:调用 remove 方法。具体实现在 __delitem__(self, key)
len(m):返回 m 中 key-val 对的个数，具体实现在 __len__(self)
key in m:返回布尔值表示 key 是否存在于 m 的 key 中。具体实在__contains__(self, key)
```

代码模板:map.py，请把代码填写到对应的位置，并在注释中对自 己的实现做出简单的解释。请在提交前做充分的测试以确认程序的 正确性。

```python
# map.py
class Map:
    def __init__(self):
        # 为方便测试，请将table大小设为100003。变量可随意添加或修改
        self.table_size = 100003
        self.slots = [None] * self.table_size
        self.data = [None] * self.table_size

    def put(self, key: int, value: int) -> None:
        pass

    def get(self, key: int) -> int:
        return -1

    def remove(self, key: int) -> None:
        pass

    def __setitem__(self, key: int, value: int) -> None:
        pass

    def __getitem__(self, key: int) -> int:
        return -1

    def __delitem__(self, key: int) -> None:
        pass

    def __len__(self) -> int:
        return 0

    def __contains__(self, key: int) -> bool:
        return False

if __name__ == '__main__':
```

```
    in_arr = input().split()
    # in_arr = "put 1 2 put 1 3 get 1 len contains 1 remove 1 len".split()
    m = Map()
    out_arr = []
    i = 0
    while i < len(in_arr):
        if in_arr[i] == 'put':
            m.put(int(in_arr[i + 1]), int(in_arr[i + 2]))
            i += 3
        elif in_arr[i] == 'get':
            value = m.get(int(in_arr[i + 1]))
            out_arr.append(str(value))
            i += 2
        elif in_arr[i] == 'remove':
            m.remove(int(in_arr[i + 1]))
            i += 2
        elif in_arr[i] == 'len':
            length = len(m)
            out_arr.append(str(length))
            i += 1
        elif in_arr[i] == 'contains':
            re = int(in_arr[i + 1]) in m
            out_arr.append(str(re))
            i += 2
    print(' '.join(out_arr))
```

可参考 Leetcode 相关题目来测试程序的正确性:

706. 设计哈希映射

- 用大小自动增长的散列表实现 ADT Set 3pts

采用开放定址冲突解决技术的散列表，课件中是固定大小的。请用 散列表实现一个 ADT Set(功能类似 python 内置的集合 set)，使 得在负载因子达到某个阈值之后，散列表的大小能自动增长。散列 表中的元素均为整数，需要实现的方法包括:

```
s=Set():创建对象，具体实现为构造函数 __init__(self)
s.add(key):将 key-value 关联加入映射
key in s:返回布尔值表示 key 是否存在于集合中。具体实现在 __contains__(self, key)
len(s):返回 s 中不同 key 的个数，具体实现在 __len__(self)
不要求:s.remove(key):从集合中删除 key，不存在则直接返回。 remove 方法在这 种实现中比较复杂，不做强制要求
```

代码模板:set.py，请把代码填写到对应的位置，并在注释中对自己 的实现做出简单的解释。请在提交前做充分的测试以确认程序的正确性。

```
# set.py
class Set:
```

```
    def __init__(self):
        # 为方便测试，请将初始table大小设为11。变量可随意添加或修改
        self.table_size = 11
        self.slots = [None] * self.table_size

    def add(self, key: int) -> None:
        pass

    def __contains__(self, key: int) -> bool:
        return False

    def __len__(self) -> int:
        return 0

    # def remove(self, key: int) -> None:
    #     pass


if __name__ == '__main__':
    in_arr = input().split()
    # in_arr = "add 1 add 2 add 1 len contains 1 contains 3".split()
    s = Set()
    out_arr = []
    i = 0
    while i < len(in_arr):
        if in_arr[i] == 'add':
            s.add(int(in_arr[i + 1]))
            i += 2
        elif in_arr[i] == 'contains':
            re = int(in_arr[i + 1]) in s
            out_arr.append(str(re))
            i += 2
        elif in_arr[i] == 'len':
            length = len(s)
            out_arr.append(str(length))
            i += 1
        elif in_arr[i] == 'remove':
            s.remove(int(in_arr[i + 1]))
            i += 2
    print(' '.join(out_arr))
```

可参考 Leetcode 相关题目来测试程序的正确性:

705. 设计哈希集合

- 快速排序 3pts

自行设计一种取“中值”的方法实现快速排序。请在注释中写一段说明，比较你的实现与课件中的快速排序算法的性能。输入为整数组成的数组，输出结果为升序排列。

代码模板: qsort.py，请把代码填写到对应的位置，并在注释中对自 己的实现做出简单的解释，并分析对比原始的快速排序算法效率的 变化。请在提交前做充分的测试以确认程序的正确性。

```python
# qsort.py
def qsort(nums: [int]) -> [int]:
    return nums

if __name__ == '__main__':
    numbers = [int(s) for s in input().split()]
    # numbers = [5, 1, 1, 2, 0, 0]
    sorted_numbers = qsort(numbers)
    print(' '.join([str(i) for i in sorted_numbers]))
```

样例输入: 5 1 1 2 0 0
样例输出: 0 0 1 1 2 5
可参考 Leetcode 相关题目来测试程序的正确性:

912. 排序数组

- 其他说明

不要求提交报告文档，但需要在注释中对实现思路有大致的说明。

leetcode 题目仅供参考，可能要求与作业不同。不要求通过leetcode，但需要测试确认自己代码已实现的功能的正确性。

请在模板代码对应的方法定义内部填写自己的程序，要求实现的的方法内部请不要产生额外的输出。

作业8 树及其算法

May 23, 2021，5月30日24 时

评分标准

	合格
输出正确表达树	分数:5 (41.66666%)输出结果是否正确
是否考虑额外的操作符	分数:2 (16.66666%)每考虑一个操作符，+0.5分
delete操作正确	分数:5 (41.66666%)操作正确

请在模版 exprTree.py.template 和 avl.py.template 的基础上，实现 exprTree.py

avl.py

- 中缀表达式解析树 3pts

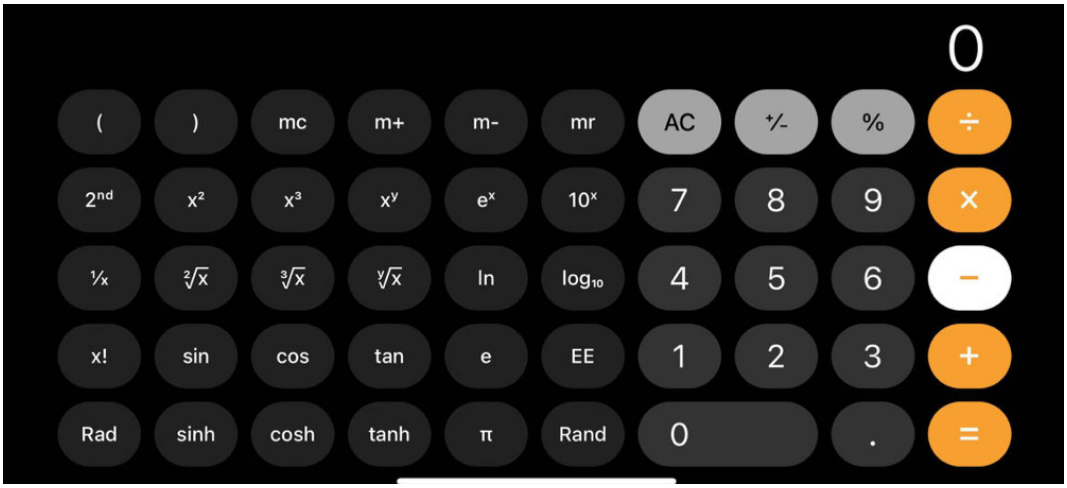
整理并扩展 buildParseTree 方法，使其能处理我们一般的中缀表达 式。

非全括号形式，支持不同操作符和括号的优先级

输入的表达式单词之间不一定有空格分隔
至少支持 “+”、“-”、“*”、“/” 这四种运算，鼓励对运算进行扩充。

运算数有多种形式，至少支持正整数和字母，鼓励按一般的习惯进行 扩充。

提交文件: exprTree.py



- 实现 AVL 树的 delete 方法 3pts

复习整理 BST 和 AVL 的代码
仿照 BST 的 detete 方法
学习 AVL 中 put 方法，在插入节点时如何保持树的平衡。

提交文件:avl.py

```
# exprTree.py.template
class BinaryTree:
    def __init__(self, rootObj, \
            left=None, right=None):              #可以同时设置节点的左右子树
        self.key = rootObj
        self.leftChild = left
        self.rightChild = right

    def insertLeft(self, newNode):
        self.leftChild = BinaryTree(newNode, \
                left=self.leftChild)

    def insertRight(self, newNode):
        self.rightChild = BinaryTree(newNode, \
                right = self.rightChild)
```

```
16
17      def getRightChild(self):
18          return self.rightChild;
19
20      def getLeftChild(self):
21          return self.leftChild
22
23      def setRootVal(self, obj):
24          self.key = obj
25
26      def getRootVal(self):
27          return self.key
28
29  def _print_t(tree, is_left, offset, depth, buf):
30      if not tree:
31          return 0
32
33      b = "{:^5}".format(tree.key)
34      width = 5
35      while len(buf)<2*depth+1:                  #让后面的buf[2*depth]访问有效
36          buf.append([])
37      left  = _print_t(tree.leftChild,  True, offset,                depth + 1,
    buf);
38      right = _print_t(tree.rightChild, False, offset + left + width, depth + 1,
    buf);
39
40      enlarge = offset+left+width+right-len(buf[2*depth])
41      buf[2*depth].extend([' ']*enlarge)
42      for i, c in enumerate(b):                  #输出子树根节点的内容
43          buf[2*depth][offset+left+i] = c
44
45
46      if depth > 0:                              #输出子树与其父节点的连线
47          if is_left:
48              enlarge = offset+left+2*width+right-len(buf[2*depth-1])
49              buf[2*depth-1].extend([' ']*enlarge)
50              for i in range(width+right):
51                  buf[2 * depth - 1][offset + left + width//2 + i] = '-'
52              buf[2 * depth - 1][offset + left + width//2] = '+';
53              buf[2 * depth - 1][offset + left + width//2 + right + width] = '+';
54          else:
55              enlarge = offset+left+width+right-len(buf[2*depth-1])
56              buf[2*depth-1].extend([' ']*enlarge)
57              for i in range(left+width):
58                  buf[2 * depth - 1][offset - width//2 + i] = '-';
59              buf[2 * depth - 1][offset + left + width//2] = '+';
60              buf[2 * depth - 1][offset - width//2 - 1] = '+';
61      return left + width + right;
62
```

```
def print_t(tree):
    buf = []
    _print_t(tree, True, 0, 0, buf)
    for l in buf:
        print(''.join(l))

def tokenlize(expr):
    """
    支持的token类型
    运算符：‘+’, ‘-’, ‘*’, ‘/’
    操作数：整数，标识符（以字母开头，包含字母或数字）
    括号：‘(’, ‘)’
    """
    for _expr in expr.split():
        for i in _tokenlize(_expr):
            yield i
def _tokenlize(expr):                       #不用管表达式中的空格了
    buf = []
    for i in expr:
        if i in '+-*/()':
            if buf:
                yield ''.join(buf); buf=[]
            yield i
        elif i.isalnum():
            if buf and buf[0].isdigit() and i.isalpha():
                yield ''.join(buf); buf=[]
            buf.append(i)
        else:
            raise ValueError("Unknown charactor:'{}'".format(i))
    if buf:
        yield ''.join(buf); buf=[]

priority = {'(':0, '+':1, '-':1, '*':2, '/':2}
def buildParseTree(inexp):                  #中缀表达式
    subtree_stack = []                      #用栈来保存中间过程中生成的子树
    op_stack = []                           #用栈来保存操作符。
                                            #请参考以前表达式中缀转后缀的代码，
    for t in tokenlize(inexp):              #从字符串中提取token
        # 这里原来有30行左右的代码，实现了从中缀表达式到二叉树结构的转换。
        # 现在它们被删掉了，请同学们重新实现。
        pass
    if len(subtree_stack) > 1:
        raise SyntaxError("Unexpected operand '{}'".format(subtree_stack[-1]))
    #最后子树栈中的唯一元素，就是我们要求的表达式树。
    return subtree_stack[0]

#下面的测试代码请不要改动，它从输入读取一行作为表达式字符串，然后完成两件事情：
#       1. 输出表达式中包括的单词
#       2. 调用buildParseTree进行二叉树转换，并把得到的表达式树打印出来。
```

```
#
if __name__ == "__main__":
    expr = input()
    #expr = "(a + b)*h/2"
    for t in tokenlize(expr):
        print(t)
    pt = buildParseTree(expr)
    print_t(pt)
```

```
# avl.py.template
def _print_t(tree, is_left, offset, depth, buf, label):
    if not tree:
        return 0

    b = "{:^5}".format(label(tree))
    width = 5
    while len(buf)<2*depth+1:                #让后面的buf[2*depth]访问有效
        buf.append([])
    left  = _print_t(tree.leftChild,  True, offset,                depth + 1, buf, label);
    right = _print_t(tree.rightChild, False, offset + left + width, depth + 1, buf, label);

    enlarge = offset+left+width+right-len(buf[2*depth])
    buf[2*depth].extend([' ']*enlarge)
    for i, c in enumerate(b):                #输出子树根节点的内容
        buf[2*depth][offset+left+i] = c


    if depth > 0:                            #输出子树与其父节点的连线
        if is_left:
            enlarge = offset+left+2*width+right-len(buf[2*depth-1])
            buf[2*depth-1].extend([' ']*enlarge)
            for i in range(width+right):
                buf[2 * depth - 1][offset + left + width//2 + i] = '-'
            buf[2 * depth - 1][offset + left + width//2] = '+';
            buf[2 * depth - 1][offset + left + width//2 + right + width] = '+';
        else:
            enlarge = offset+left+width+right-len(buf[2*depth-1])
            buf[2*depth-1].extend([' ']*enlarge)
            for i in range(left+width):
                buf[2 * depth - 1][offset - width//2 + i] = '-';
            buf[2 * depth - 1][offset + left + width//2] = '+';
            buf[2 * depth - 1][offset - width//2 - 1] = '+';
    return left + width + right;

def print_t(tree, label=lambda x:x.key):
```

```
    buf = []
    _print_t(tree, True, 0, 0, buf, label)
    for l in buf:
        print(''.join(l))

class TreeNode:
    def __init__(self, key, val, left=None,
                 right=None, parent=None):
        self.key = key
        self.payload = val
        self.leftChild = left
        self.rightChild = right
        self.parent = parent              #增加了对parent的指回
        self.balanceFactor = 0            #在BST中用不到，也没什么坏处。

    def hasLeftChild(self):               #这个函数实际没必要，只是为了代码的讲解更加口语化。
        return self.leftChild

    def hasRightChild(self):
        return self.rightChild

    def isLeftChild(self):
        return self.parent and self.parent.leftChild == self

    def isRightChild(self):
        return self.parent and self.parent.rightChild == self

    def isRoot(self):
        return not self.parent

    def hasAnyChildren(self):
        return self.rightChild or self.leftChild

    def hasBothChildren(self):
        return self.rightChild and self.leftChild

    def isLeaf(self):
        return not self.hasAnyChildren()

    def flat(self):
        flat = lambda x: flat(x.leftChild)+[x.key]+flat(x.rightChild) if x else []
        return flat(self)

    def replaceNodeData(self, key, value, lc, rc):
        self.key = key
        self.payload = value
        self.leftChild = lc
        self.rightChild = rc
```

```
85          if self.hasLeftChild():           #让左右子节点指回父节点
86              self.leftChild.parent = self
87          if self.hasRightChild():
88              self.rightChild.parent = self
89
90      def findSuccessor(self):
91          if self.hasRightChild():         #在BinarySearchTree.remove()中的唯一可能
92              return self.rightChild.findMin()
93          elif self.isRoot():
94              return None
95          elif self.isLeftChild():
96              return self.parent
97          else : #self.isRightChild()       #在parent子树中的最末
98              self.parent.rightChild = None #暂时移除自己
99              succ = self.parent.findSuccessor()
100             self.parent.rightChild = self #恢复自己
101             return succ
102
103     def findMin(self):
104         current = self
105         while current.hasLeftChild():     #往左下角
106             current = current.leftChild
107         return current
108
109     def __iter__(self):
110         if self:
111             if self.hasLeftChild():
112                 for elem in self.leftChild:
113                     yield elem
114             yield self.key
115             if self.hasRightChild():
116                 for elem in self.rightChild:
117                     yield elem
118
119 class BinarySearchTree:                 #函数太多了，增加了“树”类来管理与之有关的部分
120     def __init__(self):
121         self.root = None
122         self.size = 0
123     def length(self):
124         return self.size
125     def __len__(self):                  #供len(bst)调用
126         return self.size
127     def __iter__(self):
128         return self.root.__iter__()
129
130     def display(self):
131         print_t(self.root)
132
133     def put(self,key,val=0):
```

```
        if self.root:
            self.size += self._put(key,val,self.root)
        else:
            self.root = TreeNode(key, val)
            self.size += 1

    def _put(self, key, val, currentNode):
        if key < currentNode.key:            #递归左子树
            if currentNode.hasLeftChild():
                return self._put(key, val, currentNode.leftChild)
            else:
                currentNode.leftChild = \
                    TreeNode(key, val, parent=currentNode)
                return 1
        elif key > currentNode.key:          #递归右子树
            if currentNode.hasRightChild():
                return self._put(key, val, currentNode.rightChild)
            else:
                currentNode.rightChild = \
                    TreeNode(key, val, parent=currentNode)
                return 1
        else: #key == currentNode.key        #出现重复key
            currentNode.payload = val
            return 0

    def __setitem__(self, k, v):
        self.put(k,v)

    def get(self, key):
        if self.root:            #这里的判断和_get()内部的判断重复了
            res = self._get(key, self.root)
            if res:
                return res.payload
            else:
                return None
        else:
            return None

    def _get(self, key, currentNode):
        if not currentNode:
            return None
        elif currentNode.key == key:
            return currentNode
        elif currentNode.key > key:
            return self._get(key, currentNode.leftChild)
        else: #currentNode.key < key
            return self._get(key, currentNode.rightChild)

    def __getitem__(self, key):
```

```
        return self.get(key)

    def __contains__(self, key):
        if self._get(key, self.root):
            return True
        else:
            return False

    def delete(self, key):
        if self.size > 1:
            nodeToRemove = self._get(key, self.root)
            if nodeToRemove:
                self.remove(nodeToRemove)
                self.size = self.size - 1
            else:
                raise KeyError('Error, key not in tree')
        elif self.size == 1 and self.root.key == key:
            self.root = None           #删除根节点
            self.size = self.size - 1
        else:
            raise KeyError('Error, key not in tree')

    def __delitem__(self, key):
        self.delete(key)

    def remove(self, currentNode):
        if currentNode.isLeaf():               #作为叶节点的最简单情况
            #移除叶节点只需要在它的父节点中把它移除
            if currentNode.isLeftChild():
                currentNode.parent.leftChild = None
            elif currentNode.isRightChild():
                currentNode.parent.rightChild = None
        elif currentNode.hasBothChildren():    #有左右两个子节点的复杂情况
            succ = currentNode.findSuccessor() #前驱、后继肯定都有
            currentNode.key = succ.key
            currentNode.payload = succ.payload
            self.remove(succ)                  #succ没有左子节点，为什么？不会出现第二层递归
        else:                                  #只有一个子节点的情况
            # 取得唯一子节点，不关心左右
            child = currentNode.leftChild \
                    if currentNode.hasLeftChild() \
                    else currentNode.rightChild  #可以砍掉一半的源代码
            if currentNode.isLeftChild():     #用子节点替换当前节点
                currentNode.parent.leftChild = child
                child.parent = currentNode.parent
            elif currentNode.isRightChild():
                currentNode.parent.rightChild = child
                child.parent = currentNode.parent
```

```
            else :                                  #当前节点是根节点，直接替换成子节点
                currentNode.replaceNodeData(child.key,
                                            child.payload,
                                            child.leftChild,
                                            child.rightChild)

class AVL(BinarySearchTree):
    def _put(self, key, val, currentNode):
        if key < currentNode.key:
            if currentNode.hasLeftChild():
                return self._put(key, val, currentNode.leftChild)
            else:
                currentNode.leftChild = TreeNode(key, val, parent=currentNode)
                self.updateBalance(currentNode.leftChild)  #调整平衡因子
                return 1
        elif key > currentNode.key:
            if currentNode.hasRightChild():
                return self._put(key, val, currentNode.rightChild)
            else:
                currentNode.rightChild = TreeNode(key, val, parent=currentNode)
                self.updateBalance(currentNode.rightChild)  #调整平衡因子
                return 1
        else: #key == currentNode.key
            currentNode.payload = val          #无新增节点，平衡因子不变
            return 0

    def updateBalance(self, node):
        if node.balanceFactor > 1 or node.balanceFactor < -1: #先看自己是否要调整
            self.rebalance(node)                           #重新平衡，并且不会向上传递
            return
        if node.parent != None:                                #更新父节点平衡因子
            if node.isLeftChild():
                node.parent.balanceFactor += 1
            elif node.isRightChild():
                node.parent.balanceFactor -= 1
            if node.parent.balanceFactor != 0:                 #调整父节点平衡因子
                self.updateBalance(node.parent)                #"=0"也会阻断递归的传递

    def rotateLeft(self, rotRoot):
        newRoot = rotRoot.rightChild                          #把新根节点提上来
        rotRoot.rightChild = newRoot.leftChild                #给新根节点的左子节点重新找
位置
        if newRoot.leftChild != None:
            newRoot.leftChild.parent = rotRoot                #    并给它指定新的parent
        newRoot.parent = rotRoot.parent                       #新根节点完全取代旧根节点
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
```

```
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        newRoot.leftChild = rotRoot
        rotRoot.parent = newRoot
        #调整新、旧根节点的平衡因子，为什么这样？马上推导。
        rotRoot.balanceFactor = rotRoot.balanceFactor + \
                                1 - min(newRoot.balanceFactor, 0)
        newRoot.balanceFactor = newRoot.balanceFactor + \
                                1 + max(rotRoot.balanceFactor, 0)

    def rotateRight(self, rotRoot):
        newRoot = rotRoot.leftChild
        rotRoot.leftChild = newRoot.rightChild    #56，58新根的右子转为旧根的左子
        if newRoot.rightChild != None:
            newRoot.rightChild.parent = rotRoot
        newRoot.parent = rotRoot.parent           #59，64/66新根取代旧根和父建立关系
        if rotRoot.isRoot():
            self.root = newRoot
        else:
            if rotRoot.isLeftChild():
                rotRoot.parent.leftChild = newRoot
            else:
                rotRoot.parent.rightChild = newRoot
        newRoot.rightChild = rotRoot              #67，68旧根下沉为新根的子节点
        rotRoot.parent = newRoot
        #调整新、旧根节点的平衡因子，为什么这样？马上推导。
        rotRoot.balanceFactor = rotRoot.balanceFactor - \
                                1 - max(newRoot.balanceFactor, 0)
        newRoot.balanceFactor = newRoot.balanceFactor - \
                                1 + min(rotRoot.balanceFactor, 0)

    def rebalance(self, node):   #"<-1"或">1"才会被updateBalance调用
        if node.balanceFactor < -1:  #右重需要左旋
            if node.rightChild.balanceFactor > 0:
                # 右子节点左重，先对它进行一次右旋
                self.rotateRight(node.rightChild)
            #正常左旋
            self.rotateLeft(node)
        elif node.balanceFactor > 1:
            if node.leftChild.balanceFactor < 0:
                self.rotateLeft(node.leftChild)
            self.rotateRight(node)

    def display(self):
        print_t(self.root, label=lambda x:"{}:{}".format(x.key, x.balanceFactor))

if __name__ == "__main__":
    tree = AVL()
```

```
    """
    tree.put(1)
    tree.put(2)
    tree.display()
    tree.put(30)
    tree.put(40)
    tree.put(50)
    tree.put(60)
    tree.put(70)
    tree.put(80)
    tree.put(0)
    tree.put(25)
    tree.put(35)
    tree.put(21)
    tree.display()
    del tree[40]
    tree.display()
    """
    command = input().split()
    while command[0] != 'exit':
        if command[0] == 'put':
            tree.put(int(command[1]))
        elif command[0] == 'delete':
            print("delete()用的是BST的哦，不能保持平衡哦！")
            tree.delete(int(command[1]))
        elif command[0] == 'display':
            tree.display()
        command = input().split()
```

期末笔试复习

谢正茂 webg@PKU-Mail (北京大学计算机系)

目录

概述

基于有穷观点的能行方法

计算的基本概念

抽象计算模型-图灵机

计算复杂性及不可计算问题

不同问题的计算是有不同复杂度的，有些问题是无法计算的。

数据结构和抽象数据类型

抽象的概念，及什么是 ADT 和 DS(三.1)

算法及衡量算法

有穷、确定、可行 (一.6)

算法基本概念和衡量算法的方法

…… . …. …… …

算法分析

程序与算法的关系

算法分析的概念

计算资源及资源消耗指标，运行时间检测方法

算法复杂度的衡量指标: 大 O 表示法

确定大 O 的方法及常见的大 O 数量级函数 (二.1)

O(1), O(log(n)), O(n), O(n ∗ log(n)), O(n2), O(2n)

对算法的实现代码进行分析，以得到大 O 数量级 (二.2)

理解常见数据类型中操作的大 O 数量级

…… . …. …… …

基本数据结构

线性数据结构的概念，理解 ADT 的不同实现方案及其复杂度分析 栈的概念、特性和 ADT Stack
队列的概念、特性和 ADT Queue(一.3)
双端队列的概念、特性和 ADT Deque(二.4)

列表的概念、特性和 ADT List、ADT OrderedList

…… . …. …… …

递归与动态规划

递归的概念及初步例子
递归的“三定律”
用递归解决进制转换问题
递归调用的内部实现: 与栈相关

递归与自相似图形，理解绘制自相似图形的递归算法

用递归解决河内塔问题和探索迷宫问题
动态规划算法策略

从兑换硬币问题对比递归算法和动态规划算法，如何避免递归爆炸

…… . …. …… …

搜索算法

顺序搜索算法，以及在无序表和有序表数据结构中的不同实现
二分搜索算法，分而治之的算法策略

高效算法的额外开销问题，以及依据实际应用来选择算法

散列的概念，及散列冲突概念，完美散列函数

散列函数设计的几种方法 (一.7/8/9)

散列冲突解决方案，负载因子

开放定址法: 线性探测

数据链法

抽象数据类型 ADT Map 及实现的算法分析

…… . …. …… …

排序算法

冒泡排序算法，及性能改进
选择排序算法 (多趟比对，但减少交换次数)
插入排序算法 (为“新项”寻找插入位置，逐步扩大已排序子列表)

谢尔排序算法 (固定间隔的多个子列表进行插入排序，减小间隔)

归并排序算法 (将列表持续分裂为两半后，再合并完成排序)

快速排序算法 (以中值作为基准将列表分为小于和大于两部分)

根据数据特征来选择排序算法

算法性能退化

如何选择一个好的算法来解决问题

…… . …. …… …

排序算法

(一.10/11/12/13/14，二.9)

什么时候出现最好情况，什么时候出现最差情况?

复杂度主要讨论比较次数
稳定性的定义是什么?

适用场景有什么限制?

算法名称	最好情况	最差情况	平均情况	稳定性
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$	稳定
选择排序	$O(n)$	$O(n^2)$	$O(n^2)$	不稳定
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$	稳定
谢尔排序	$O(n)$	$O(n^2)$	$O(n^{\frac{3}{2}})$	不稳定
归并排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	稳定
快速排序	$O(nlog(n))$	$O(n^2)$	$O(nlog(n))$	不稳定
分配排序	$O(d(m+n))$	$O(d(m+n))$	$O(d(m+n))$	稳定
堆排序	$O(nlog(n))$	$O(nlog(n))$	$O(nlog(n))$	不稳定

…… . …. …… …

树及其算法

树的概念及例子，树的两种定义

　熟悉树相关的术语与定义

实现树的方法: 嵌套列表法、节点链接法

树的应用: 解析树 (语法树和表达式树)

表达式树的建立算法，利用表达式解析树求值

树的遍历: 前序、中序及后序遍历 (二.3/7)

　在表达式生成和求值中的应用

Huffman 树编码

优先队列的概念，完全二叉树与二叉堆 (一.5，二.5/6/8)

二叉搜索树 BST 及平衡树 AVL 树的概念及实现 (一.18，二.11，四.2)

树(林)与二叉树的互相转化

…… . …. …… …

图及其算法

图的概念，用图来表示的网络

熟悉图的术语及定义，ADT Graph

图的实现方法: 邻接矩阵及邻接列表法

词梯问题及广度优先搜索 BFS

骑士周游问题及深度优先搜索 DFS

通用的深度优先搜索算法 (一.16)

DFS 用于解决拓扑排序和强连通分支问题 (一.15/17，三.2)

路由选择，最短路径问题及 Dijkstra 算法 (一.19)

信息广播，最小生成树问题及 Prim 算法 (二.10)

…… . …. …… …

期末闭卷笔试

时间:6 月 22 日(周二) 下午 1:00 点-3:00 点

地点: 理教 406

百分制，占总评百分之四十

…… . …. …… …

References

[1]. Problem Solving with Algorithms and Data Structures Using Python

https://runestone.academy/ns/books/published//pythonds/index.html

https://github.com/RunestoneInteractive/pythonds

[2]. Introduction to Algorithms Third Edition, Thomas H.Cormen, Charles E.Leiserson, Ronald L.Rivest, Clifford Stein. The MIT Press

[3]. 数据结构与算法（B），谢正茂，2021年春季.