Assignment #8: 🌲 (2/3)

Updated 2223 GMT+8 Oct 27, 2025

2025 fall, Complied by 林奕妃、环境科学与工程学院

> **说明：**
>
> 1. **解题与记录：**
>
>   对于每一个题目，请提供其解题思路（可选），并附上使用Python或C++编写的源代码（确保已在OpenJudge， Codeforces，LeetCode等平台上获得Accepted）。请将这些信息连同显示“Accepted”的截图一起填写到下方的作业模板中。（推荐使用Typora https://typoraio.cn 进行编辑，当然你也可以选择Word。）无论题目是否已通过，请标明每个题目大致花费的时间。
>
> 2. **提交安排：**提交时，请首先上传PDF格式的文件，并将.md或.doc格式的文件作为附件上传至右侧的“作业评论”区。确保你的Canvas账户有一个清晰可见的本人头像，提交的文件为PDF格式，并且“作业评论”区包含上传的.md或.doc附件。
>
> 3. **延迟提交：**如果你预计无法在截止日期前提交作业，请提前告知具体原因。这有助于我们了解情况并可能为你提供适当的延期或其他帮助。
>
> 请按照上述指导认真准备和提交作业，以保证顺利完成课程要求。

1. 题目

E108.将有序数组转换为二叉搜索树（30分钟）

https://leetcode.cn/problems/convert-sorted-array-to-binary-search-tree/

思路：升序序列为左→根→右的元素排列为从小到大，选取数组的中间元素作为当前树（或子树）的根节点。采用分治+递归的方法处理，对于将问题简化为排列后选择中间的元素作为根，再分别对左右支进一步进行转换。

代码：

```
from typing import List, Optional

class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        """
        主函数，通过调用递归辅助函数来构建平衡二叉搜索树。
        """
        # 初始调用，传入整个数组的索引范围。
        return self._build_recursive(nums, 0, len(nums) - 1)

    def _build_recursive(self, nums: List[int], left: int, right: int) ->
Optional[TreeNode]:
        """
```

```
        使用递归和分治法，根据数组的指定范围 [left, right] 构建树。
        """
        # 递归终止条件：当左指针越过右指针，表示当前区间为空。
        if left > right:
            return None

        # 找到区间的中间点，作为当前子树的根节点。
        # 这样可以保证左右子树的节点数量差不超过1，从而使树平衡。
        mid = (left + right) // 2

        # 直接使用平台提供的 TreeNode 类创建节点。
        root = TreeNode(nums[mid])

        # 递归构建左右子树。
        # - 左子树由中间点左边的元素构成。
        root.left = self._build_recursive(nums, left, mid - 1)
        # - 右子树由中间点右边的元素构成。
        root.right = self._build_recursive(nums, mid + 1, right)

        # 返回构建好的当前子树的根节点。
        return root
```

代码运行截图 （至少包含有"Accepted"）

M07161: 森林的带度数层次序列存储（30分钟）

tree, http://cs101.openjudge.cn/practice/07161/

思路：数据是按照广度优先（BFS）的顺序给出的，每一个字母后面给出其子树的数量；再用深度优先（DFS）的方法进行后跟遍历。

代码：

```
import sys
from collections import deque

# 1. 定义树的节点结构
# 每个节点有一个值和一个子节点列表
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
```

```
# 2. 实现重建树的函数
def reconstruct_tree(data_list):
    """根据带度数的层次序列，重建一棵树。"""
    if not data_list:
        return None

    # 创建根节点
    root_value, root_degree = data_list[0]
    root = TreeNode(root_value)

    # 队列中存放 (节点对象, 该节点的度数)
    # 这是实现层次重建的关键数据结构
    queue = deque([(root, root_degree)])

    # 索引，指向下一个可用的孩子节点
    child_index = 1

    while queue:
        parent_node, parent_degree = queue.popleft()

        # 为父节点连接其所有孩子
        for _ in range(parent_degree):
            if child_index >= len(data_list):
                break  # 防止数据格式错误导致索引越界

            child_value, child_degree = data_list[child_index]
            child_node = TreeNode(child_value)

            parent_node.children.append(child_node)
            queue.append((child_node, child_degree))

            child_index += 1

    return root

# 3. 实现后根遍历的函数
def post_order_traversal(node, result_list):
    """对树进行后根遍历 (先子后根)，并将结果存入列表。"""
    if not node:
        return

    # 递归遍历所有子节点
    for child in node.children:
        post_order_traversal(child, result_list)

    # 最后访问节点本身
    result_list.append(node.value)

# 4. 主逻辑
def main():
    """
    程序主入口，负责读取输入、调用功能函数并打印输出。
    """
    lines = sys.stdin.readlines()
    if not lines:
        return
```

```
    # 第一行是树的数量 n
    # 我们其实可以不使用 n，直接处理后面的行即可
    # n = int(lines[0].strip())

    total_post_order_result = []

    # 从第二行开始，每一行都是一棵树的数据
    for line in lines[1:]:
        line = line.strip()
        if not line:
            continue

        parts = line.split()

        # 将输入行解析成 (节点值, 度数) 的元组列表
        degree_level_data = []
        # 使用 try-except 块增加代码的健壮性
        try:
            for i in range(0, len(parts), 2):
                value = parts[i]
                degree = int(parts[i+1])
                degree_level_data.append((value, degree))
        except (ValueError, IndexError):
            # 如果数据格式错误，跳过这一行
            continue

        # a. 重建树
        root = reconstruct_tree(degree_level_data)

        # b. 对重建好的树进行后根遍历
        tree_result = []
        post_order_traversal(root, tree_result)

        # c. 将这棵树的结果追加到总结果中
        total_post_order_result.extend(tree_result)

    # 关键修正：使用 " ".join() 来确保输出字符间有空格
    print(" ".join(total_post_order_result))

if __name__ == "__main__":
    main()
```

代码运行截图 （至少包含有"Accepted"）

#50677494提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys
from collections import deque

# 1. 定义树的节点结构
# 每个节点有一个值和一个子节点列表
class TreeNode:
    def __init__(self, value):
        self.value = value
```

基本信息

#: 50677494
题目: 07161
提交人: 25n2200013554
内存: 3704kB
时间: 21ms
语言: Python3
提交时间: 2025-11-02 21:46:46

M27928: 遍历树（30分钟）

adjacency list, dfs, http://cs101.openjudge.cn/practice/27928/

思路：输入格式是典型的邻接表表示法，即每一行列出了一个节点和它的所有直接子节点。这个遍历方式的核心在于先处理值小的，再处理值大的，这个规则应用在当前节点和其直接子节点这个局部范围内。注意题目没有直接给出哪个是根节点，因此没有父节点的就是根节点，需要我们进行筛选。

代码：

```
import sys

# 1. 定义树的节点结构
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

# 2. 实现自定义的遍历函数
def custom_traverse(node):
    """
    根据题目定义的特殊规则进行递归遍历。
    """
    if not node:
        return

    # a. 收集当前节点和它的所有子节点
    items_to_sort = [node] + node.children

    # b. 根据节点的值进行排序
    #    使用 lambda 函数来指定排序的依据是节点的 value 属性
    items_to_sort.sort(key=lambda x: x.value)

    # c. 遍历排序后的列表
    for item in items_to_sort:
        if item == node:
            # 如果是父节点本身，直接打印值
            print(item.value)
        else:
            # 如果是子节点，则递归调用遍历函数
            custom_traverse(item)

# 3. 主逻辑函数
def solve():
    """
    负责读取输入、构建树、找到根节点并启动遍历。
    """
    # 读取所有输入行
    lines = sys.stdin.readlines()
    if len(lines) < 2:
        return
```

```
    # n = int(lines[0]) # 节点数，在这里我们也可以不使用它

    # --- 树的构建 ---
    nodes = {}           # 字典：存储所有节点对象，键为节点值
    child_values = set() # 集合：存储所有出现过的子节点的值

    for line in lines[1:]:
        # 解析每一行，得到数值列表
        parts = list(map(int, line.strip().split()))
        if not parts:
            continue

        parent_val = parts[0]

        # 确保父节点对象存在
        if parent_val not in nodes:
            nodes[parent_val] = TreeNode(parent_val)
        parent_node = nodes[parent_val]

        # 处理所有子节点
        for child_val in parts[1:]:
            # 记录这是一个子节点
            child_values.add(child_val)

            # 确保子节点对象存在
            if child_val not in nodes:
                nodes[child_val] = TreeNode(child_val)
            child_node = nodes[child_val]

            # 建立父子关系
            parent_node.children.append(child_node)

    # --- 寻找根节点 ---
    # 根节点就是那个在所有节点中，但从未当过子节点的节点
    root_val = -1
    for val in nodes:
        if val not in child_values:
            root_val = val
            break

    # 如果找到了根节点，则从根节点开始遍历
    if root_val != -1:
        root_node = nodes[root_val]
        custom_traverse(root_node)

# 程序入口
if __name__ == "__main__":
    # 为了防止递归深度过大，在某些平台上可能需要增加递归深度限制
    # sys.setrecursionlimit(1000)
    solve()
```

代码运行截图 （至少包含有"Accepted"）

#50677895提交状态

查看 提交 统计 提问

状态：Accepted

源代码

```
import sys

# 1. 定义树的节点结构
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []
```

基本信息

#: 50677895
题目: 27928
提交人: 25n2200013554
内存: 3840kB
时间: 21ms
语言: Python3
提交时间: 2025-11-02 22:16:15

M129.求根节点到叶节点数字之和（30分钟）

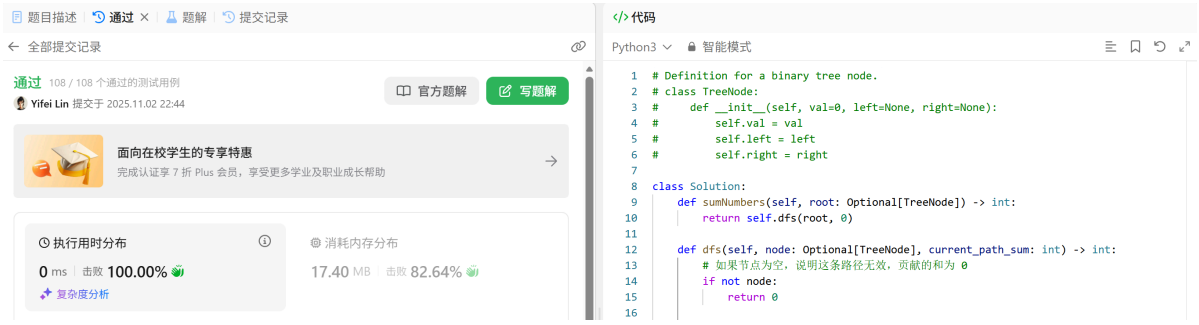
dfs, https://leetcode.cn/problems/sum-root-to-leaf-numbers/

思路：使用深度优先搜索，找到叶节点后返回这条路径的数字加和。

代码

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def sumNumbers(self, root: Optional[TreeNode]) -> int:
        return self.dfs(root, 0)

    def dfs(self, node: Optional[TreeNode], current_path_sum: int) -> int:
        # 如果节点为空，说明这条路径无效，贡献的和为 0
        if not node:
            return 0

        # 计算到达当前节点为止的路径数字
        current_path_sum = current_path_sum * 10 + node.val

        # 如果是叶子节点，说明一条完整路径已形成，返回这条路径的值
        if not node.left and not node.right:
            return current_path_sum

        # 如果不是叶子节点，返回其左右子树下所有路径数字之和
        left_sum = self.dfs(node.left, current_path_sum)
        right_sum = self.dfs(node.right, current_path_sum)

        return left_sum + right_sum
```

代码运行截图（至少包含有"Accepted"）



M24729: 括号嵌套树（30分钟）

dfs, stack, http://cs101.openjudge.cn/practice/24729/

思路： 使用栈来解析括号嵌套字符串，并构建树。遇到 '('，将当前节点作为父节点压入栈，遇到 ')'，一个父节点的子节点列表处理完毕，出栈。

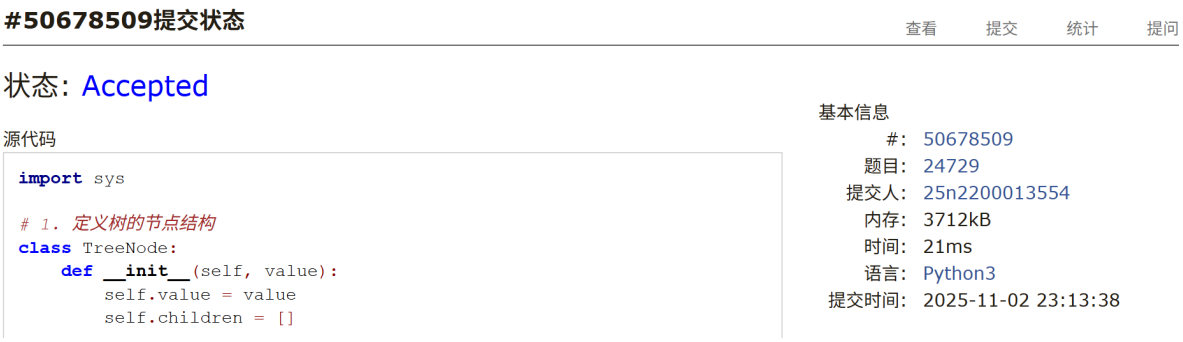
代码

```
import sys

# 1. 定义树的节点结构
class TreeNode:
    def __init__(self, value):
        self.value = value
        self.children = []

# 2. 解析字符串并构建树
def build_tree_from_string(s):
    """
    使用栈来解析括号嵌套字符串，并构建树。
    """
    if not s:
        return None

    stack = []
    root = None
    current_node = None

    for char in s:
        if 'A' <= char <= 'Z':
            # 遇到字母，创建一个新节点
            node = TreeNode(char)
            if not root:
                # 第一个遇到的节点是根节点
                root = node

            if stack:
                # 如果栈不为空，新节点是栈顶节点的子节点
                parent = stack[-1]
                parent.children.append(node)

            current_node = node
```

```
        elif char == '(':
            # 遇到 '(', 将当前节点作为父节点压入栈
            stack.append(current_node)

        elif char == ')':
            # 遇到 ')', 一个父节点的子节点列表处理完毕，出栈
            stack.pop()

        # 逗号 ',' 直接忽略

    return root

# 3. 实现前序遍历
def pre_order_traversal(node, result):
    """
    前序遍历：根 -> 子
    """
    if not node:
        return

    # 首先访问根节点
    result.append(node.value)

    # 然后依次递归遍历所有子节点
    for child in node.children:
        pre_order_traversal(child, result)

# 4. 实现后序遍历
def post_order_traversal(node, result):
    """
    后序遍历：子 -> 根
    """
    if not node:
        return

    # 首先依次递归遍历所有子节点
    for child in node.children:
        post_order_traversal(child, result)

    # 最后访问根节点
    result.append(node.value)

# 5. 主逻辑
def solve():
    """
    读取输入，调用函数，打印输出。
    """
    # 读取一行输入
    input_str = sys.stdin.readline().strip()

    # a. 根据输入构建树
    root = build_tree_from_string(input_str)

    # b. 执行前序遍历
    pre_order_result = []
```

```
    pre_order_traversal(root, pre_order_result)
    print("".join(pre_order_result))

    # c. 执行后序遍历
    post_order_result = []
    post_order_traversal(root, post_order_result)
    print("".join(post_order_result))

# 程序入口
if __name__ == "__main__":
    solve()
```

代码运行截图（至少包含有"Accepted"）



T02775: 文件结构“图”（1个半小时）

tree, http://cs101.openjudge.cn/practice/02775/

思路：用递归函数来处理嵌套的目录结构，用“树节点”来表示文件和目录，再用另一个递归函数按格式、排序规则把树打印出来。

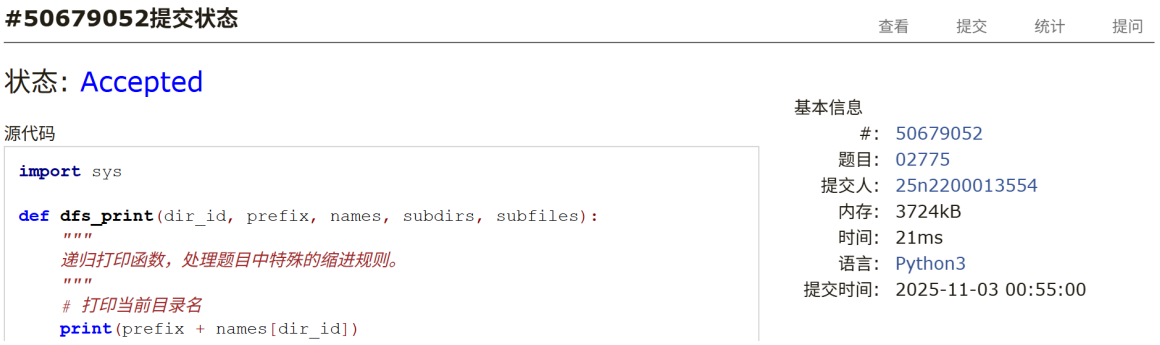
代码：

```
import sys

def dfs_print(dir_id, prefix, names, subdirs, subfiles):
    """
    递归打印函数，处理题目中特殊的缩进规则。
    """
    # 打印当前目录名
    print(prefix + names[dir_id])

    # 子目录的缩进要更深一级
    subdir_prefix = prefix + "|     "
    for subdir_id in subdirs[dir_id]:
        dfs_print(subdir_id, subdir_prefix, names, subdirs, subfiles)

    # 文件的缩进与父目录名相同
    for file_name in sorted(subfiles[dir_id]):
        print(prefix + file_name)
```

```python
def build_tree():
    """
    读取一组测试数据（直到 '*' 或 '#'），构建树结构。
    返回一个元组 (data, status)，其中：
    - data: 如果读取到内容，则是包含树信息的元组 (names, subdirs, subfiles)；否则是None。
    - status: 'OK' 表示以 '*' 结束，'END' 表示以 '#' 或文件末尾结束。
    """
    names = ["ROOT"]
    subdirs = [[]]
    subfiles = [[]]
    next_id = 1
    stack = [0]

    has_content = False

    for line in sys.stdin:
        line = line.strip()
        if not line:
            continue

        if line == '*':
            # 正常结束一个数据集
            return (names, subdirs, subfiles), 'OK'

        if line == '#':
            # 整个输入结束
            # 如果'#'前有数据，也算一个数据集
            if has_content:
                return (names, subdirs, subfiles), 'END'
            else:
                return None, 'END'

        has_content = True
        if line.startswith('d'):
            names.append(line)
            subdirs.append([])
            subfiles.append([])
            subdirs[stack[-1]].append(next_id)
            stack.append(next_id)
            next_id += 1
        elif line == ']':
            stack.pop()
        else: # isfile
            subfiles[stack[-1]].append(line)

    # 处理文件末尾（EOF）的情况
    if has_content:
        return (names, subdirs, subfiles), 'END'
    else:
        return None, 'END'


def main():
    """
    主函数，精确控制数据集之间的空行。
```

```
    """
    cas = 1
    while True:
        data, status = build_tree()

        if data:
            # 如果不是第一个数据集，则在打印前输出一个空行
            if cas > 1:
                print()

            print(f"DATA SET {cas}:")
            names, subdirs, subfiles = data
            dfs_print(0, "", names, subdirs, subfiles)
            cas += 1

        # 如果状态是 'END'，无论有没有数据，都应终止循环
        if status == 'END':
            break

if __name__ == "__main__":
    main()
```

代码运行截图 （至少包含有"Accepted"）



#50679052提交状态

查看 提交 统计 提问

状态: Accepted

源代码

```
import sys

def dfs_print(dir_id, prefix, names, subdirs, subfiles):
    """
    递归打印函数，处理题目中特殊的缩进规则。
    """
    # 打印当前目录名
    print(prefix + names[dir_id])
```

基本信息

#: 50679052
题目: 02775
提交人: 25n2200013554
内存: 3724kB
时间: 21ms
语言: Python3
提交时间: 2025-11-03 00:55:00

2. 学习总结和个人收获

感觉树的概念更加抽象了，且这次作业不像之前的二叉树作业，出现了3个及以上的子树，题目变的更加复杂，同时感觉题干变的更难理解了，有的题画图辅助理解起来也有点困难，可能我对于树的理解还不够到位，因此还需要进一步再巩固一下。尤其是M27928: 遍历树的sample2，一开始先入为主，没有想到在第一行给出的10不是根节点，居然是后给出的2的子节点，导致对整个题目的理解都有困难。
T02775: 文件结构“图”这个编程理解起来也很有困难。