Assignment #6: "树"算: Huffman,BinHeap,BST,AVL,DisjointSet

Updated 2214 GMT+8 March 24, 2024

2024 spring, Complied by ~~天幂~~ 化学与分子工程学院

说明:

1）这次作业内容不简单，耗时长的话直接参考题解。

2）请把每个题目解题思路（可选），源码Python, 或者C++（已经在Codeforces/Openjudge上AC），截图（包含Accepted），填写到下面作业模版中（推荐使用 typora https://typoraio.cn ，或者用word）。AC 或者没有AC，都请标上每个题目大致花费时间。

3）提交时候先提交pdf文件，再把md或者doc文件上传到右侧“作业评论”。Canvas需要有同学清晰头像、提交文件有pdf、"作业评论"区有上传的md或者doc附件。

4）如果不能在截止前提交作业，请写明原因。

编程环境

操作系统：Windows 11 23H2

Python编程环境：Visual Studio Code 1.86.2230.

1. 题目

22275: 二叉搜索树的遍历

http://cs101.openjudge.cn/practice/22275/

思路：利用搜索树的性质，直接排序得到中序表达式，再套用22158的代码。

代码

```
from copy import copy
class Node(object):
    _ID = 0
    NodeID:int
    pNodeID:int
    name:str
    sub:list    #List<Node>
    depth:int
    def __init__(self, name, l, pNodeID= -1, depth:int=0):
        self.NodeID = self._ID
```

```
        self.__class__._ID += 1
        self.pNodeID = pNodeID
        self.sub = []
        self.name = name
        self.depth = depth
        for node in l:
            self.sub.append(node)
    def info(self):
        return (self.NodeID, self.sub)

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID, cSubNodes = node.info()

        self.tree[cNodeID] = node   #加入树

        if not self.root:   #尝试转移根节点
            self.root = node
        elif self.get(self.root.NodeID) in cSubNodes:
            self.root = node

        for nodes in self.tree.values():    #尝试添加父节点
            aNodeID, aSubNodes = nodes.info()
            if self.get(cNodeID) in aSubNodes:  #是子节点
                node.pNodeID = aNodeID
    def get(self, nodeID):
        if nodeID == -1:
            return False
        else:
            return self.tree[nodeID]
    def getDepth(self, node:Node):
        cSubNodes = node.sub
        if cSubNodes:
            if node.depth == 0:
                self.depth = 1 + max([self.getDepth(subNode) for subNode in
cSubNodes])
            return node.depth
        return 0
    def getTreeDep(self):   # This can also init the tree
        return self.getDepth(self.root)

    def levelOrderFrom(self, node:Node):
        if not node: return []

        res, queue = [], [node]
        while queue:
            level_node = []

            for _ in range(len(queue)):
                node = queue.pop(0)
```

```
                level_node.append(node.name)

                for x in node.sub:
                    if x:
                        queue.append(x)
            res.append(level_node)

        return "".join(["".join(x) for x in res])

    def levelOrder(self):
         return self.levelOrderFrom(self.root)
    def preOrderFrom(self, node:Node):  # 先序遍历
        if not node: return ''
        return node.name + " " + "".join([self.preOrderFrom(x) for x in node.sub])
    def preOrder(self):
        return self.preOrderFrom(self.root)
    def postOrderFrom(self, node:Node):  # 后序遍历
        if not node: return ''
        return  "".join([self.postOrderFrom(x) for x in node.sub]) + " " + node.name
    def postOrder(self):
        return self.postOrderFrom(self.root)




def toTree(preOrPost: str, middle: str, index: int) -> Tree:
    def toNode(tree: Tree, middle: str, preOrPost: str, index: int):
        try:
            rootName = preOrPost[index]
            rootIndex = middle.index(rootName)
            info = middle[:rootIndex], middle[rootIndex + 1:], preOrPost[1:rootIndex + 1], preOrPost[rootIndex + 1:]
        except IndexError:
            return False
        if info == ('', '', '', ''):
            node = Node(rootName, [])
            tree.add(node)
            return(node)
        lSubTreeMiddle, rSubTreeMiddle, lSubTreePreOrPost, rSubTreePreOrPost = info
        node = Node(rootName, [toNode(tree, lSubTreeMiddle, lSubTreePreOrPost, index), toNode(tree, rSubTreeMiddle, rSubTreePreOrPost, index)])
        tree.add(node)
        return(node)
    myTree = Tree()
    toNode(myTree, middle, preOrPost, index)
    return myTree

_, pre = input(), list(input().split())
middle = pre.copy()
middle.sort(key=int)
print(toTree(pre, middle, 0).postOrder().lstrip())
```

代码运行截图

状态: Accepted

源代码

```
from copy import copy
class Node(object):
    _ID = 0
    NodeID:int
    pNodeID:int
    name:str
    sub:list    #List<Node>
    depth:int
    def __init__(self, name, l, pNodeID= -1, depth:int=0):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.pNodeID = pNodeID
        self.sub = []
        self.name = name
        self.depth = depth
        for node in l:
            self.sub.append(node)
    def info(self):
        return (self.NodeID, self.sub)

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID, cSubNodes = node.info()

        self.tree[cNodeID] = node    #加入树

        if not self.root:    #尝试转移根节点
            self.root = node
        elif self.get(self.root.NodeID) in cSubNodes:
            self.root = node

        for nodes in self.tree.values():    #尝试添加父节点
            aNodeID, aSubNodes = nodes.info()
            if self.get(cNodeID) in aSubNodes:  #是子节点
                node.pNodeID = aNodeID
    def get(self, nodeID):
        if nodeID == -1:
            return False
        else:
            return self.tree[nodeID]
    def getDepth(self, node:Node):
        cSubNodes = node.sub
        if cSubNodes:
            if node.depth == 0:
                self.depth = 1 + max([self.getDepth(subNode) for subNode
            return node.depth
        return 0
    def getTreeDep(self):    # This can also init the tree
        return self.getDepth(self.root)
```

```
    def levelOrderFrom(self, node:Node):
        if not node: return []

        res, queue = [], [node]
        while queue:
            level_node = []

            for _ in range(len(queue)):
                node = queue.pop(0)
                level_node.append(node.name)

                for x in node.sub:
                    if x:
                        queue.append(x)
            res.append(level_node)

        return "".join(["".join(x) for x in res])

    def levelOrder(self):
         return self.levelOrderFrom(self.root)
    def preOrderFrom(self, node:Node):  # 先序遍历
        if not node: return ''
        return node.name + " " + "".join([self.preOrderFrom(x) for x in
    def preOrder(self):
        return self.preOrderFrom(self.root)
    def postOrderFrom(self, node:Node):  # 后序遍历
        if not node: return ''
        return  "".join([self.postOrderFrom(x) for x in node.sub]) + " "
    def postOrder(self):
        return self.postOrderFrom(self.root)



def toTree(preOrPost: str, middle: str, index: int) -> Tree:
    def toNode(tree: Tree, middle: str, preOrPost: str, index: int):
        try:
            rootName = preOrPost[index]
            rootIndex = middle.index(rootName)
            info = middle[:rootIndex], middle[rootIndex + 1:], preOrPost
        except IndexError:
            return False
        if info == ('', '', '', ''):
            node = Node(rootName, [])
            tree.add(node)
            return(node)
        lSubTreeMiddle, rSubTreeMiddle, lSubTreePreOrPost, rSubTreePreO
        node = Node(rootName, [toNode(tree, lSubTreeMiddle, lSubTreePre
        tree.add(node)
        return(node)
    myTree = Tree()
    toNode(myTree, middle, preOrPost, index)
    return myTree

_, pre = input(), list(input().split())
middle = pre.copy()
middle.sort(key=int)
print(toTree(pre, middle, 0).postOrder().lstrip())
```

05455: 二叉搜索树的层次遍历

http://cs101.openjudge.cn/practice/05455/

思路：难点在于概念理解（？）看了题解才明白题目的意思。

代码

```
class Node(object):
    _ID = 0
    NodeID:int
    name:str
    sub:list    #List<Node>
    def __init__(self, name, l):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.sub = []
        self.name = name
        self.sub = l

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID = node.NodeID

        self.tree[cNodeID] = node   #加入树

        if not self.root:   #尝试转移根节点
            self.root = node

    def levelOrderFrom(self, node:Node):
        if not node: return []

        res, queue = [], [node]
        while queue:
            level_node = []

            for _ in range(len(queue)):
                node = queue.pop(0)
                level_node.append(node.name)

                for x in node.sub:
                    if x:
                        queue.append(x)
```

```
            res.append(level_node)

        return " ".join([" ".join(x) for x in res])

    def levelOrder(self):
         return self.levelOrderFrom(self.root)

def compare(a:str, b:str) -> bool:
    return int(a) < int(b)

def toNode(root:Node, node:Node) -> None:
    if not root:
        return node
    if compare(node.name, root.name):
        root.sub[0] = toNode(root.sub[0], node)
    else:
        root.sub[1] = toNode(root.sub[1], node)
    return root

def toTree(l:list) -> Tree:
    myTree = Tree()
    root = None
    for string in l:
        node = Node(string, [False, False])
        myTree.add(node)
        root = toNode(root, node)
    return myTree

l = list(input().split())
l = list(dict.fromkeys(l))
print(toTree(l).levelOrder())
```

代码运行截图

状态: Accepted

源代码

```
class Node(object):
    _ID = 0
    NodeID:int
    name:str
    sub:list    #List<Node>
    def __init__(self, name, l):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.sub = []
        self.name = name
        self.sub = l

class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):
        cNodeID = node.NodeID

        self.tree[cNodeID] = node    #加入树

        if not self.root:    #尝试转移根节点
            self.root = node

    def levelOrderFrom(self, node:Node):
        if not node: return []

        res, queue = [], [node]
        while queue:
            level_node = []

            for _ in range(len(queue)):
                node = queue.pop(0)
                level_node.append(node.name)

                for x in node.sub:
                    if x:
                        queue.append(x)
            res.append(level_node)

        return " ".join([" ".join(x) for x in res])

    def levelOrder(self):
         return self.levelOrderFrom(self.root)

def compare(a:str, b:str) -> bool:
    return int(a) < int(b)

def toNode(root:Node, node:Node) -> None:
    if not root:
        return node
    if compare(node.name, root.name):
        root.sub[0] = toNode(root.sub[0], node)
```

```
        root.sub[0] = toNode(root.sub[0], node)
    else:
        root.sub[1] = toNode(root.sub[1], node)
    return root

def toTree(l:list) -> Tree:
    myTree = Tree()
    root = None
    for string in l:
        node = Node(string, [False, False])
        myTree.add(node)
        root = toNode(root, node)
    return myTree

l = list(input().split())
l = list(dict.fromkeys(l))
print(toTree(l).levelOrder())
```


04078: 实现堆结构

http://cs101.openjudge.cn/practice/04078/

练习自己写个BinHeap。当然机考时候，如果遇到这样题目，直接import heapq。手搓栈、队列、堆、AVL等，考试前需要搓个遍。

思路：继承list类简化代码。主要是sinkdown要注意，因为不能保证左边比右边小，需要判断。

代码

```
class BinHeap(list):
    def insert(self, __object) -> None:
        super().append(__object)
        index = len(self) - 1
        self._siftup(index)
    def pop(self):
        x = self[-1]
        del(self[-1])
        if self:
            self[0] = x
            self._sinkdown(0)

    def _swap(self, index1, index2) -> None:
            self[index1], self[index2] = self[index2], self[index1]
    def _siftup(self, index) -> None:
        pIndex = (index - 1)//2
        if pIndex >= 0 and self[pIndex] > self[index]:
            self._swap(pIndex, index)
            return self._siftup(pIndex)
        return
```

```
    def _sinkdown(self, index) -> None:
        lIndex = 2 * index + 1
        rIndex = lIndex + 1
        s = self[index]
        try:
            l = self[lIndex]
            mIndex = lIndex
            try:
                r = self[rIndex]
                if r < l:
                    mIndex = rIndex
            except IndexError:
                pass
            if self[mIndex] < s:
                self._swap(mIndex, index)
                return self._sinkdown(mIndex)
        except IndexError:
            pass
        return

    def popAndReturn(self) -> int:
        x = self[0]
        self.pop()
        return x
    def op(self, op, element=None):
        if op == 1:
            self.insert(element)
        else:
            print(self.popAndReturn())

n = int(input())
myBinHeap = BinHeap()
for _ in range(n):
    myBinHeap.op(*map(int, input().split()))
```

代码运行截图

状态: Accepted

源代码

```
class BinHeap(list):
    def insert(self, __object) -> None:
        super().append(__object)
        index = len(self) - 1
        self._siftup(index)
    def pop(self):
        x = self[-1]
        del(self[-1])
        if self:
            self[0] = x
            self._sinkdown(0)

    def _swap(self, index1, index2) -> None:
            self[index1], self[index2] = self[index2], self[index1]
    def _siftup(self, index) -> None:
        pIndex = (index - 1)//2
        if pIndex >= 0 and self[pIndex] > self[index]:
            self._swap(pIndex, index)
            return self._siftup(pIndex)
        return
    def _sinkdown(self, index) -> None:
        lIndex = 2 * index + 1
        rIndex = lIndex + 1
        s = self[index]
        try:
            l = self[lIndex]
            mIndex = lIndex
            try:
                r = self[rIndex]
                if r < l:
                    mIndex = rIndex
            except IndexError:
                pass
            if self[mIndex] < s:
                self._swap(mIndex, index)
                return self._sinkdown(mIndex)
        except IndexError:
            pass
        return

    def popAndReturn(self) -> int:
        x = self[0]
        self.pop()
        return x
    def op(self, op, element=None):
        if op == 1:
            self.insert(element)
        else:
            print(self.popAndReturn())

n = int(input())
myBinHeap = BinHeap()
for _ in range(n):
    myBinHeap.op(*map(int, input().split()))
```

22161: 哈夫曼编码树

http://cs101.openjudge.cn/practice/22161/

思路：树的构建不难，只是参数多。数字转字母和字母转数字用了不同的方法解决。

代码

```
import heapq

class Node(object):
    _ID = 0
    NodeID:int
    name:str
    sub:list    #List<Node>
    weight:int
    code:str
    strl:list
    def __init__(self, name, weight, l, strl):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.name = name
        self.sub = l
        self.weight=weight
        self.code = ''
        self.strl = strl

    def __lt__(self, other):
        if self.weight < other.weight:
            return True
        elif self.weight == other.weight:
            if self.name != '':
                return self.name < other.name
            else:
                return min(self.strl) < min(other.strl)
        return False

    def getName(self):
        if self.name != '':
            return self.name
        else:
            return None

    def updateCode(self, i: str) -> None:
        if self.name != '':
            self.code = i + self.code
        for subNode in self.sub:
            if subNode:
                subNode.updateCode(i)
```

```
42
43
44  class Tree(object):
45      tree:dict
46      root:Node
47      def __init__(self):
48          self.tree = dict()
49          self.root = None
50
51      def add(self, node:Node):
52
53          self.tree[node.NodeID] = node    #加入树
54          if node.name != '':
55              self.tree[node.name] = node
56
57          if not self.root:    #尝试转移根节点
58              self.root = node
59
60  def toTree(l: list) -> Tree:
61      myTree = Tree()
62      for node in l:
63          myTree.add(node)
64      while len(l) > 1:
65          x, y =  heapq.heappop(l), heapq.heappop(l)
66          x.updateCode('0'); y.updateCode('1')
67          neoNode = Node('', x.weight + y.weight, [x, y], x.strl + y.strl +
    [x.getName(), y.getName()])
68          myTree.add(neoNode)
69          heapq.heappush(l, neoNode)
70      node = l[0]
71      myTree.root = node
72      return myTree
73
74
75  def convert(string: str, myTree: Tree) -> str:
76      try:
77          _ = int(string)
78          root = myTree.root
79          res = ''
80          i = 0
81          while i < len(string):
82              char = string[i]
83              x = root.sub[int(char)]
84              if x:
85                  root = x
86                  i += 1
87              else:
88                  res += root.name
89                  root = myTree.root
90          return res + root.name
91
92      except ValueError:
93          res = ''
94          for char in string:
95              res += myTree.tree[char].code
96          return res
```

```
97
98  l = []
99  for _ in range(int(input())):
100     n, w = input().split()
101     heapq.heappush(l, Node(n, int(w), [False, False], []))
102 myTree = toTree(l)
103 while 1:
104     try:
105         print(convert(input(), myTree))
106     except EOFError:
107         break
```

代码运行截图

状态: Accepted

源代码

```
import heapq

class Node(object):
    _ID = 0
    NodeID:int
    name:str
    sub:list    #List<Node>
    weight:int
    code:str
    strl:list
    def __init__(self, name, weight, l, strl):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.name = name
        self.sub = l
        self.weight=weight
        self.code = ''
        self.strl = strl

    def __lt__(self, other):
        if self.weight < other.weight:
            return True
        elif self.weight == other.weight:
            if self.name != '':
                return self.name < other.name
            else:
                return min(self.strl) < min(other.strl)
        return False

    def getName(self):
        if self.name != '':
            return self.name
        else:
            return None

    def updateCode(self, i: str) -> None:
        if self.name != '':
            self.code = i + self.code
        for subNode in self.sub:
            if subNode:
                subNode.updateCode(i)


class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None

    def add(self, node:Node):

        self.tree[node.NodeID] = node    #加入树
        if node.name != '':
            self.tree[node.name] = node
```

```
        if not self.root:    #尝试转移根节点
            self.root = node

def toTree(l: list) -> Tree:
    myTree = Tree()
    for node in l:
        myTree.add(node)
    while len(l) > 1:
        x, y =  heapq.heappop(l), heapq.heappop(l)
        x.updateCode('0'); y.updateCode('1')
        neoNode = Node('', x.weight + y.weight, [x, y], x.strl + y.strl
        myTree.add(neoNode)
        heapq.heappush(l, neoNode)
    node = l[0]
    myTree.root = node
    return myTree


def convert(string: str, myTree: Tree) -> str:
    try:
        _ = int(string)
        root = myTree.root
        res = ''
        i = 0
        while i < len(string):
            char = string[i]
            x = root.sub[int(char)]
            if x:
                root = x
                i += 1
            else:
                res += root.name
                root = myTree.root
        return res + root.name

    except ValueError:
        res = ''
        for char in string:
            res += myTree.tree[char].code
        return res

l = []
for _ in range(int(input())):
    n, w = input().split()
    heapq.heappush(l, Node(n, int(w), [False, False], []))
myTree = toTree(l)
while 1:
    try:
        print(convert(input(), myTree))
    except EOFError:
        break
```

晴问9.5: 平衡二叉树的建立

https://sunnywhy.com/sfbj/9/5/359

思路：最关键的点在于insert函数的格式，最开始用的空返回值，想了很久没能写出来。最后看了一眼题解发现改一下返回值逻辑就很顺畅。

代码

```
class Node(object):
    _ID = 0
    NodeID:int
    name:str
    value:int
    height:int
    sub:list

    def __init__(self, name, l):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.name = name
        self.value = int(name)
        self.sub = l
        self.height = 0
    def getHeight(self):
        x = max([-1] + [subNode.getHeight() for subNode in self.sub if
subNode]) + 1
        self.height = x
        return x
    def getBF(self):
        left = right = -1
        if self.sub[0]:
            left = self.sub[0].getHeight()
        if self.sub[1]:
            right = self.sub[1].getHeight()
        return left - right


class Tree(object):
    tree:dict
    root:Node
    def __init__(self):
        self.tree = dict()
        self.root = None
    def insert(self, node) -> None:
        if self.root == None:
            self.root = node
        else:
            self.root = self._insert(self.root, node)
    def _insert(self, root, node:Node) -> None:
        if not root:
            return node
```

```
        else:
            if node.value < root.value:
                root.sub[0] = self._insert(root.sub[0], node)
            else:
                root.sub[1] = self._insert(root.sub[1], node)

            bf = root.getBF()
            if bf == 2:
                if node.value > root.sub[0].value:
                   root.sub[0] = self._rotate(root.sub[0], True)
                return self._rotate(root, False)
            elif bf == -2:
                if node.value < root.sub[1].value:
                    root.sub[1] = self._rotate(root.sub[1], False)
                return self._rotate(root, True)
            return root
    def _rotate(self, node:Node, lRotate:bool):
        bNode = node.sub[lRotate]
        dNode = bNode.sub[not lRotate]
        if dNode:
            dNode.pNode = node
        node.sub[lRotate] = dNode
        bNode.sub[not lRotate] = node
        return bNode

    def preOrderFrom(self, node:Node):
        return node.name + " " + "".join([self.preOrderFrom(x) for x in
node.sub if x])
    def preOrder(self):
        return self.preOrderFrom(self.root)


myTree = Tree()
_ = input()
for x in input().split():
    myTree.insert(Node(x, [False, False]))
print(myTree.preOrder().rstrip())
```

代码运行截图

代码书写

Python

```python
class Node(object):
    _ID = 0
    NodeID:int
    name:str
    value:int
    height:int
    sub:list

    def __init__(self, name, l):
        self.NodeID = self._ID
        self.__class__._ID += 1
        self.name = name
        self.value = int(name)
        self.sub = l
        self.height = 0
    def getHeight(self):
        x = max([-1] + [subNode.getHeight() for subNode in self.sub if
        self.height = x
        return x
    def getBF(self):
        left = right = -1
        if self.sub[0]:
            left = self.sub[0].getHeight()
        if self.sub[1]:
            right = self.sub[1].getHeight()
        return left - right
```

测试输入 提交结果 历史提交

完美通过 查看题解

100% 数据通过测试

运行时长: 0 ms

收起面板 运行 提交

02524: 宗教信仰

http://cs101.openjudge.cn/practice/02524/

思路：最基本的并查集，没什么好说的。

代码

```python
class DisjointSet(object):
    father_dict:dict
    def __init__(self, l):
        self.father_dict = {}
```

```
        for x in l:
            self.father_dict[x] = x
    def find(self, x):
        if self.father_dict[x] == x:
            return x
        else:
            return self.find(self.father_dict[x])
    def union(self, x, y):
        px = self.find(x)
        py = self.find(y)
        if px != py:
            self.father_dict[py] = px
i = 0
while 1:
    try:
        n, m = map(int, input().split())
        if n == m == 0: raise EOFError
        i += 1
        ds = DisjointSet(list(range(1, n + 1)))
        for _ in range(m):
            a, b = map(int, input().split())
            if a > b:
                a, b = b, a
            ds.union(a, b)
        rgs = set(ds.find(x) for x in range(1, n + 1))
        print("Case {}: {}".format(i, len(rgs)))
    except EOFError:
        break
```

代码运行截图

状态：Accepted

源代码

```
class DisjointSet(object):
    father_dict:dict
    def __init__(self, l):
        self.father_dict = {}
        for x in l:
            self.father_dict[x] = x
    def find(self, x):
        if self.father_dict[x] == x:
            return x
        else:
            return self.find(self.father_dict[x])
    def union(self, x, y):
        px = self.find(x)
        py = self.find(y)
        if px != py:
            self.father_dict[py] =px
i = 0
while 1:
    try:
        n, m = map(int, input().split())
        if n == m == 0: raise EOFError
        i += 1
        ds = DisjointSet(list(range(1, n + 1)))
        for _ in range(m):
            a, b = map(int, input().split())
            if a > b:
                a, b = b, a
            ds.union(a, b)
        rgs = set(ds.find(x) for x in range(1, n + 1))
        print("Case {}: {}".format(i, len(rgs)))
    except EOFError:
        break
```


2. 学习总结和收获

感觉有时候需要具体分析一个方法该怎么写，虽然两种方式在一般情况下都能达成目的，但是不同场景下需求不同，就会有好用和不好用的区别。