

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

Updated 2214 GMT+8 March 24, 2024

2024 spring, Compiled by 天幕 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

```
1  from copy import copy
2  class Node(object):
3      _ID = 0
4      NodeID:int
5      pNodeID:int
6      name:str
7      sub:list    #List<Node>
8      depth:int
9      def __init__(self, name, l, pNodeID= -1, depth:int=0):
10         self.NodeID = self._ID
```

```

11         self.__class__._ID += 1
12         self.pNodeID = pNodeID
13         self.sub = []
14         self.name = name
15         self.depth = depth
16         for node in l:
17             self.sub.append(node)
18     def info(self):
19         return (self.NodeID, self.sub)
20
21 class Tree(object):
22     tree:dict
23     root:Node
24     def __init__(self):
25         self.tree = dict()
26         self.root = None
27
28     def add(self, node:Node):
29         cNodeID, cSubNodes = node.info()
30
31         self.tree[cNodeID] = node    #加入树
32
33         if not self.root:    #尝试转移根节点
34             self.root = node
35         elif self.get(self.root.NodeID) in cSubNodes:
36             self.root = node
37
38         for nodes in self.tree.values():    #尝试添加父节点
39             aNodeID, aSubNodes = nodes.info()
40             if self.get(cNodeID) in aSubNodes:    #是子节点
41                 node.pNodeID = aNodeID
42     def get(self, nodeID):
43         if nodeID == -1:
44             return False
45         else:
46             return self.tree[nodeID]
47     def getDepth(self, node:Node):
48         cSubNodes = node.sub
49         if cSubNodes:
50             if node.depth == 0:
51                 self.depth = 1 + max([self.getDepth(subNode) for subNode in
cSubNodes])
52             return node.depth
53         return 0
54     def getTreeDep(self):    # This can also init the tree
55         return self.getDepth(self.root)
56
57     def levelOrderFrom(self, node:Node):
58         if not node: return []
59
60         res, queue = [], [node]
61         while queue:
62             level_node = []
63
64             for _ in range(len(queue)):
65                 node = queue.pop(0)

```

```

66         level_node.append(node.name)
67
68         for x in node.sub:
69             if x:
70                 queue.append(x)
71         res.append(level_node)
72
73         return "".join(["".join(x) for x in res])
74
75     def levelOrder(self):
76         return self.levelOrderFrom(self.root)
77     def preOrderFrom(self, node:Node): # 先序遍历
78         if not node: return ''
79         return node.name + " " + "".join([self.preOrderFrom(x) for x in
node.sub])
80     def preOrder(self):
81         return self.preOrderFrom(self.root)
82     def postOrderFrom(self, node:Node): # 后序遍历
83         if not node: return ''
84         return "".join([self.postOrderFrom(x) for x in node.sub]) + " " +
node.name
85     def postOrder(self):
86         return self.postOrderFrom(self.root)
87
88
89
90
91 def toTree(preOrPost: str, middle: str, index: int) -> Tree:
92     def toNode(tree: Tree, middle: str, preOrPost: str, index: int):
93         try:
94             rootName = preOrPost[index]
95             rootIndex = middle.index(rootName)
96             info = middle[:rootIndex], middle[rootIndex + 1:],
preOrPost[1:rootIndex + 1], preOrPost[rootIndex + 1:]
97         except IndexError:
98             return False
99         if info == ('', '', '', ''):
100             node = Node(rootName, [])
101             tree.add(node)
102             return(node)
103         lSubTreeMiddle, rSubTreeMiddle, lSubTreePreOrPost,
rSubTreePreOrPost = info
104         node = Node(rootName, [toNode(tree, lSubTreeMiddle,
lSubTreePreOrPost, index), toNode(tree, rSubTreeMiddle, rSubTreePreOrPost,
index)])
105         tree.add(node)
106         return(node)
107     myTree = Tree()
108     toNode(myTree, middle, preOrPost, index)
109     return myTree
110
111 _, pre = input(), list(input().split())
112 middle = pre.copy()
113 middle.sort(key=int)
114 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 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]) + " "
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[index + 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, 0),
                               toNode(tree, rSubTreeMiddle, rSubTreePreOrPost, 0)])
        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

```
1 class Node(object):
2     _ID = 0
3     NodeID:int
4     name:str
5     sub:list    #List<Node>
6     def __init__(self, name, l):
7         self.NodeID = self._ID
8         self.__class__._ID += 1
9         self.sub = []
10        self.name = name
11        self.sub = l
12
13    class Tree(object):
14        tree:dict
15        root:Node
16        def __init__(self):
17            self.tree = dict()
18            self.root = None
19
20        def add(self, node:Node):
21            cNodeID = node.NodeID
22
23            self.tree[cNodeID] = node    #加入树
24
25            if not self.root:    #尝试转移根节点
26                self.root = node
27
28        def levelOrderFrom(self, node:Node):
29            if not node: return []
30
31            res, queue = [], [node]
32            while queue:
33                level_node = []
34
35                for _ in range(len(queue)):
36                    node = queue.pop(0)
37                    level_node.append(node.name)
38
39                    for x in node.sub:
40                        if x:
41                            queue.append(x)
```

```

42         res.append(level_node)
43
44         return " ".join([" ".join(x) for x in res])
45
46     def levelOrder(self):
47         return self.levelOrderFrom(self.root)
48
49 def compare(a:str, b:str) -> bool:
50     return int(a) < int(b)
51
52 def toNode(root:Node, node:Node) -> None:
53     if not root:
54         return node
55     if compare(node.name, root.name):
56         root.sub[0] = toNode(root.sub[0], node)
57     else:
58         root.sub[1] = toNode(root.sub[1], node)
59     return root
60
61 def toTree(l:list) -> Tree:
62     myTree = Tree()
63     root = None
64     for string in l:
65         node = Node(string, [False, False])
66         myTree.add(node)
67         root = toNode(root, node)
68     return myTree
69
70 l = list(input().split())
71 l = list(dict.fromkeys(l))
72 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())

```

©2002-2022 POJ 京ICP备20010980号-1

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

```

1 class BinHeap(list):
2     def insert(self, __object) -> None:
3         super().append(__object)
4         index = len(self) - 1
5         self._siftup(index)
6     def pop(self):
7         x = self[-1]
8         del(self[-1])
9         if self:
10             self[0] = x
11             self._sinkdown(0)
12
13     def _swap(self, index1, index2) -> None:
14         self[index1], self[index2] = self[index2], self[index1]
15     def _siftup(self, index) -> None:
16         pIndex = (index - 1) // 2
17         if pIndex >= 0 and self[pIndex] > self[index]:
18             self._swap(pIndex, index)
19             return self._siftup(pIndex)
20     return

```

```

21     def _sinkdown(self, index) -> None:
22         lIndex = 2 * index + 1
23         rIndex = lIndex + 1
24         s = self[index]
25         try:
26             l = self[lIndex]
27             mIndex = lIndex
28             try:
29                 r = self[rIndex]
30                 if r < l:
31                     mIndex = rIndex
32             except IndexError:
33                 pass
34             if self[mIndex] < s:
35                 self._swap(mIndex, index)
36                 return self._sinkdown(mIndex)
37         except IndexError:
38             pass
39         return
40
41     def popAndReturn(self) -> int:
42         x = self[0]
43         self.pop()
44         return x
45     def op(self, op, element=None):
46         if op == 1:
47             self.insert(element)
48         else:
49             print(self.popAndReturn())
50
51 n = int(input())
52 myBinHeap = BinHeap()
53 for _ in range(n):
54     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

```
1 import heapq
2
3 class Node(object):
4     _ID = 0
5     NodeID:int
6     name:str
7     sub:list    #List<Node>
8     weight:int
9     code:str
10    strl:list
11    def __init__(self, name, weight, l, strl):
12        self.NodeID = self._ID
13        self.__class__._ID += 1
14        self.name = name
15        self.sub = l
16        self.weight=weight
17        self.code = ''
18        self.strl = strl
19
20    def __lt__(self, other):
21        if self.weight < other.weight:
22            return True
23        elif self.weight == other.weight:
24            if self.name != '':
25                return self.name < other.name
26            else:
27                return min(self.strl) < min(other.strl)
28        return False
29
30    def getName(self):
31        if self.name != '':
32            return self.name
33        else:
34            return None
35
36    def updateCode(self, i: str) -> None:
37        if self.name != '':
38            self.code = i + self.code
39        for subNode in self.sub:
40            if subNode:
41                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

```
1 class Node(object):
2     _ID = 0
3     NodeID:int
4     name:str
5     value:int
6     height:int
7     sub:list
8
9     def __init__(self, name, l):
10         self.NodeID = self._ID
11         self.__class__._ID += 1
12         self.name = name
13         self.value = int(name)
14         self.sub = l
15         self.height = 0
16     def getHeight(self):
17         x = max([-1] + [subNode.getHeight() for subNode in self.sub if
subNode]) + 1
18         self.height = x
19         return x
20     def getBF(self):
21         left = right = -1
22         if self.sub[0]:
23             left = self.sub[0].getHeight()
24         if self.sub[1]:
25             right = self.sub[1].getHeight()
26         return left - right
27
28
29 class Tree(object):
30     tree:dict
31     root:Node
32     def __init__(self):
33         self.tree = dict()
34         self.root = None
35     def insert(self, node) -> None:
36         if self.root == None:
37             self.root = node
38         else:
39             self.root = self._insert(self.root, node)
40     def _insert(self, root, node:Node) -> None:
41         if not root:
42             return node
```

```

43         else:
44             if node.value < root.value:
45                 root.sub[0] = self._insert(root.sub[0], node)
46             else:
47                 root.sub[1] = self._insert(root.sub[1], node)
48
49         bf = root.getBF()
50         if bf == 2:
51             if node.value > root.sub[0].value:
52                 root.sub[0] = self._rotate(root.sub[0], True)
53             return self._rotate(root, False)
54         elif bf == -2:
55             if node.value < root.sub[1].value:
56                 root.sub[1] = self._rotate(root.sub[1], False)
57             return self._rotate(root, True)
58         return root
59     def _rotate(self, node:Node, lRotate:bool):
60         bNode = node.sub[lRotate]
61         dNode = bNode.sub[not lRotate]
62         if dNode:
63             dNode.pNode = node
64             node.sub[lRotate] = dNode
65             bNode.sub[not lRotate] = node
66         return bNode
67
68     def preOrderFrom(self, node:Node):
69         return node.name + " " + "".join([self.preOrderFrom(x) for x in
node.sub if x])
70     def preOrder(self):
71         return self.preOrderFrom(self.root)
72
73
74 myTree = Tree()
75 _ = input()
76 for x in input().split():
77     myTree.insert(Node(x, [False, False]))
78 print(myTree.preOrder().rstrip())

```

代码运行截图

```
1 class Node(object):
2     _ID = 0
3     NodeID:int
4     name:str
5     value:int
6     height:int
7     sub:list
8
9     def __init__(self, name, l):
10         self.NodeID = self._ID
11         self.__class__._ID += 1
12         self.name = name
13         self.value = int(name)
14         self.sub = l
15         self.height = 0
16     def getHeight(self):
17         x = max([-1] + [subNode.getHeight() for subNode in self.sub] if
18         self.height = x
19         return x
20     def getBF(self):
21         left = right = -1
22         if self.sub[0]:
23             left = self.sub[0].getHeight()
24         if self.sub[1]:
25             right = self.sub[1].getHeight()
26         return left - right
27
```

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

```
1 class DisjointSet(object):
2     father_dict:dict
3     def __init__(self, l):
4         self.father_dict = {}
```

```

5         for x in l:
6             self.father_dict[x] = x
7     def find(self, x):
8         if self.father_dict[x] == x:
9             return x
10        else:
11            return self.find(self.father_dict[x])
12    def union(self, x, y):
13        px = self.find(x)
14        py = self.find(y)
15        if px != py:
16            self.father_dict[py] = px
17    i = 0
18    while 1:
19        try:
20            n, m = map(int, input().split())
21            if n == m == 0: raise EOFError
22            i += 1
23            ds = DisjointSet(list(range(1, n + 1)))
24            for _ in range(m):
25                a, b = map(int, input().split())
26                if a > b:
27                    a, b = b, a
28                ds.union(a, b)
29            rgs = set(ds.find(x) for x in range(1, n + 1))
30            print("Case {}: {}".format(i, len(rgs)))
31        except EOFError:
32            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
```

©2002-2022 POJ 京ICP备20010980号-1

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。