算法讨论班第 15 期——李玥珮

2015 年 12 月 4 号

105、

Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:

You may assume that duplicates do not exist in the tree.

解题思路：

考虑下面的例子：

- 中根遍历: D B E A F C
- 先根遍历: A B D E C F

由先根遍历序列得到，这棵树的根节点是 A，而 A 节点排在中根遍历的第四位，也就是说 A 之前的三个节点（D B E）都在节点 A 的左子树上；同理，F、C 在 A 的右子树上。如下图所示：

```
              A
            /   \
          /       \
        D B E      F C
```

此时的问题变为两个子问题，对于 A 的左子树，其

- 中根遍历：D B E
- 先根遍历：B D E

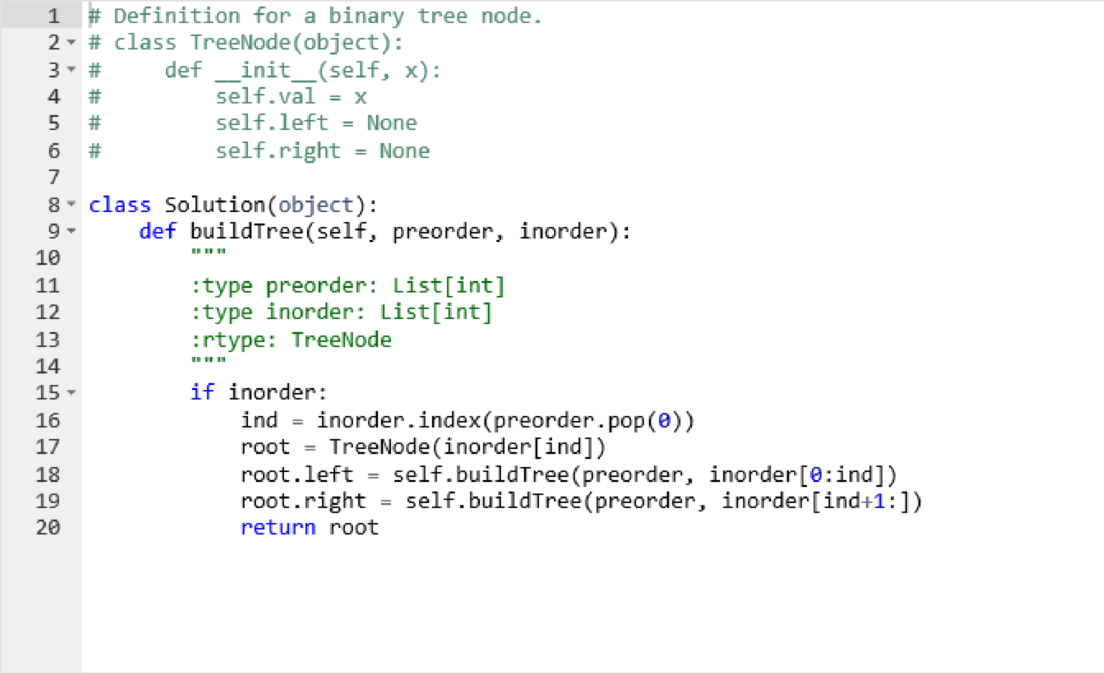
对于 A 的右子树

- 中根遍历：F C
- 先跟遍历：C F

由此可见，A 的左右子树遇到的问题与总问题是完全一样的，我们可以采用递归的思想求解，分别再求出左右子树的根节点，结果如下。我们可以这样一层层的求解，知道得到最终结果。

```
         A
       /   \
     /       \
    B          C
   / \        /
 /     \    /
D       E  F
```

代码：



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

class Solution(object):
    def buildTree(self, preorder, inorder):
        """
        :type preorder: List[int]
        :type inorder: List[int]
        :rtype: TreeNode
        """
        if inorder:
            ind = inorder.index(preorder.pop(0))
            root = TreeNode(inorder[ind])
            root.left = self.buildTree(preorder, inorder[0:ind])
            root.right = self.buildTree(preorder, inorder[ind+1:])
            return root
```

106、

Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

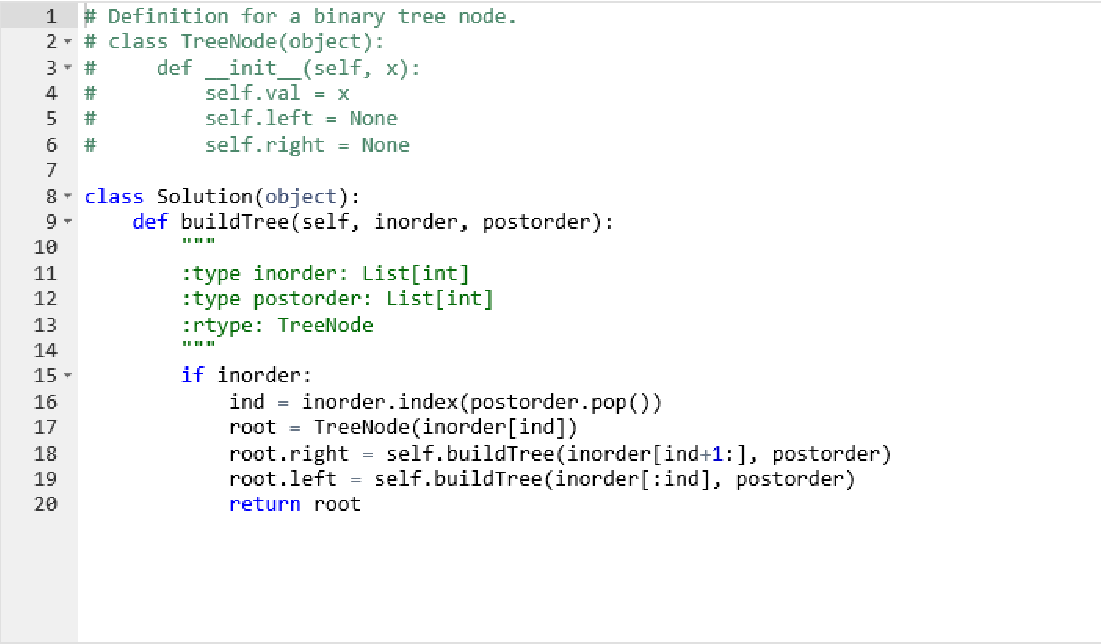
Note:

You may assume that duplicates do not exist in the tree.

结题思路：

本题与上一题类似，不同的是把先根遍历换成了后根遍历。但在算法中，它们起到的作用是一样的，都是确定递归中每层子树的根节点是什么。先根遍历的第一个节点就是根节点，而后跟遍历的最后一个节点是根节点。

代码如下：



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

class Solution(object):
    def buildTree(self, inorder, postorder):
        """
        :type inorder: List[int]
        :type postorder: List[int]
        :rtype: TreeNode
        """
        if inorder:
            ind = inorder.index(postorder.pop())
            root = TreeNode(inorder[ind])
            root.right = self.buildTree(inorder[ind+1:], postorder)
            root.left = self.buildTree(inorder[:ind], postorder)
            return root
```

133、

Clone Graph

Clone an undirected graph. Each node in the graph contains a `label` and a list of its `neighbors`.

OJ's undirected graph serialization:

Nodes are labeled uniquely.

We use `#` as a separator for each node, and `,` as a separator for node label and each neighbor of the node.

As an example, consider the serialized graph `{0,1,2#1,2#2,2}`.

The graph has a total of three nodes, and therefore contains three parts as separated by `#`.

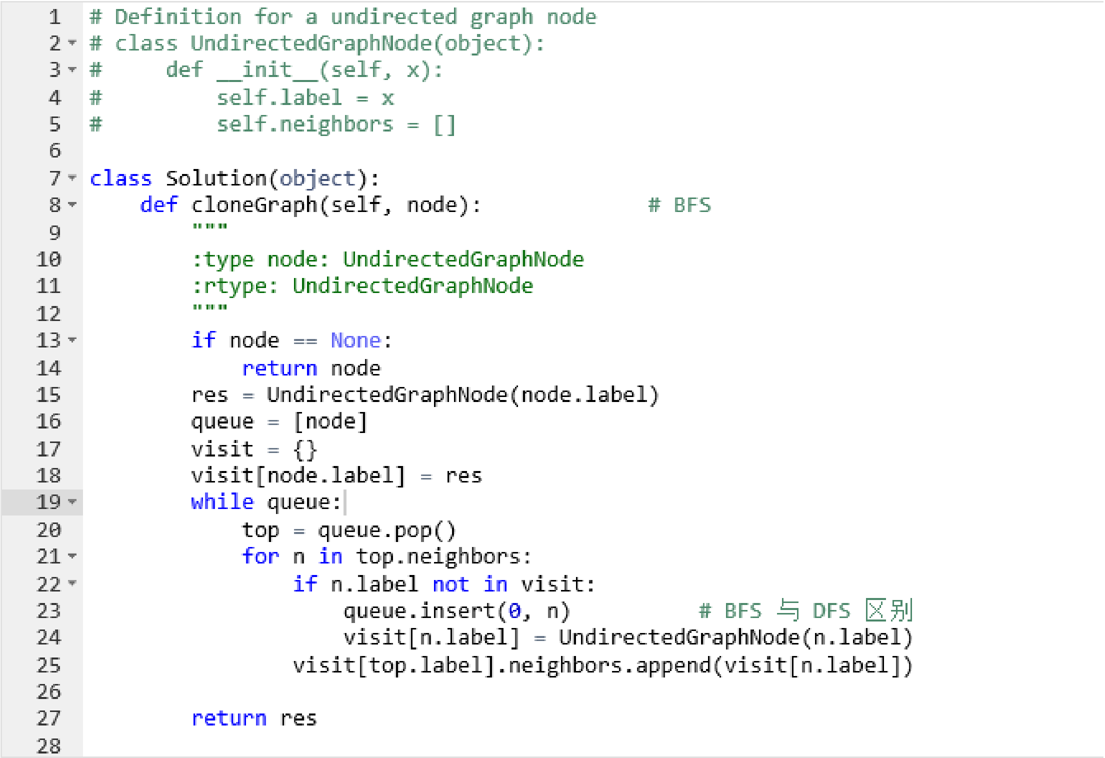
1. First node is labeled as `0`. Connect node `0` to both nodes `1` and `2`.
2. Second node is labeled as `1`. Connect node `1` to node `2`.
3. Third node is labeled as `2`. Connect node `2` to node `2` (itself), thus forming a self-cycle.

Visually, the graph looks like the following:

```
       1
      / \
     /   \
    0 --- 2
         / \
         \_/
```

结题思路：

遍历图，遍历的过程中将节点的信息与图的结构拷贝下来。遍历图的方法有DFS 与 BFS，对于本例都是用。我用的是 BFS，利用一个队列辅助遍历。代码如下：



```python
# Definition for a undirected graph node
# class UndirectedGraphNode(object):
#     def __init__(self, x):
#         self.label = x
#         self.neighbors = []

class Solution(object):
    def cloneGraph(self, node):             # BFS
        """
        :type node: UndirectedGraphNode
        :rtype: UndirectedGraphNode
        """
        if node == None:
            return node
        res = UndirectedGraphNode(node.label)
        queue = [node]
        visit = {}
        visit[node.label] = res
        while queue:
            top = queue.pop()
            for n in top.neighbors:
                if n.label not in visit:
                    queue.insert(0, n)          # BFS 与 DFS 区别
                    visit[n.label] = UndirectedGraphNode(n.label)
                visit[top.label].neighbors.append(visit[n.label])

        return res

```

问题：这种存储结构，怎么处理非连通图？