# Binary Tree

## Preorder

144. Binary Tree Preorder Traversal

### Idea: root, left, right

### 迭代就要有个Stack 不停往左下压栈，左下Node为None之后一个一个pop栈顶Node，往右下再找。

Iterative Solution

```
def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res, stack = [], []
        while root or stack:
            if root:
                res.append(root.val)
                stack.append(root)
                root = root.left
            else:
                root = stack.pop().right

        return res
```

Recursive Solution

```
def preorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []
        return [root.val] + self.preorderTraversal(root.left) +
self.preorderTraversal(root.right)
```

## Inorder

94. Binary Tree Inorder Traversal

### Idea: left, root, right

Iterative Solution

**画个图就清楚了 要先无脑往左走 压栈 走不动了 先pop最底下的 然后加进result list里面 之后再往右边找**

```
def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res, stack = [], []
        while root or stack:
            if root:
                stack.append(root)
                root = root.left
            else:
                root = stack.pop()
                res.append(root.val)
                root = root.right
        return res
```

Recursive Solution

```
def inorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:
            return []

        return self.inorderTraversal(root.left) + [root.val] +
self.inorderTraversal(root.right)
```

# Postorder

145. Binary Tree Postoder Traversal

Iterative Solution

```
def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        res, stack = [], []
        while root or stack:
            if root:
                # keep to right
                res.append(root.val)
                stack.append(root)
                root = root.right
            else:
                root = stack.pop().left

        return res[::-1]
```

Recursive Solution

```
def postorderTraversal(self, root: Optional[TreeNode]) -> List[int]:
        if not root:    return []
        return self.postorderTraversal(root.left) + self.postorderTraversal(root.right) +
[root.val]
```

---

## Level Order Traversal -> BFS, Queue 层序遍历

102. Binary Tree Level Order Traversal

### Idea: 用Queue存每一层的Node List存每一层Node的left、right *value*， 生成时先把root存进queue

Iterative Solution

```
def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        res = []
        if not root: return res
        q = collections.deque()
        q.append(root)
        while q:
            temp_res = []
            for _ in range(len(q)):
                cur = q.popleft()
                temp_res.append(cur.val)
                if cur.left:
                    q.append(cur.left)
                if cur.right:
                    q.append(cur.right)

            res.append(temp_res)
        return res
```

Recursive Solution

```
def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        # recursive, use dfs
        res = []

        def dfs(root, depth):
```

```
            # depth means the level number in Binary Tree
            if not root: return []
            if len(res) == depth: res.append([])
            res[depth].append(root.val)

            if root.left: dfs(root.left, depth + 1)
            if root.right: dfs(root.right, depth + 1)

        dfs(root, 0)
        return res
```

## Level Order Traversal 2

107. Binary Tree Level Order Traversal II

### Idea: Just same with Level Order Traverse 1, just need to reverse the list at last

Iterative Solution

```
def levelOrderBottom(self, root: Optional[TreeNode]) -> List[List[int]]:
        # level order traversal, reverse list at last
        res = []
        if not root: return res
        q = collections.deque([root])
        while q:
            tmp_res = []
            for _ in range(len(q)):
                cur = q.popleft()
                tmp_res.append(cur.val)
                if cur.left:
                    q.append(cur.left)
                if cur.right:
                    q.append(cur.right)
            res.append(tmp_res)

        return res[::-1]
```

199. Binary Tree Right Side View

### Idea: Find all right view means that every last node.val in corresponding level

Iterative Solution

```
def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        # bfs but only add right val (which is the last node value on that level)
        res = []
```

```
        if not root: return res
        q = collections.deque([root])
        while q:
            # give a variable on len(q) since the size of queue will change
            size = len(q)
            for i in range(size):
                cur = q.popleft()
                if cur.left:
                    q.append(cur.left)
                if cur.right:
                    q.append(cur.right)
                if i == size - 1:
                    res.append(cur.val)
        return res
```

Recursive Solution

```
...
```

637. Average of Levels in Binary Tree

### Idea: Find every level node.val, put in a list, get the average by doing sum(lst) / len(lst)

Iterative Solution

```
def averageOfLevels(self, root: Optional[TreeNode]) -> List[float]:
        # get the sum of level node values and get the average( sum(lst) / len(lst) )

        res = []
        if not root: return res
        q = collections.deque([root])
        while q:
            level_lst = []
            for _ in range(len(q)):
                cur = q.popleft()
                level_lst.append(cur.val)
                if cur.left:
                    q.append(cur.left)
                if cur.right:
                    q.append(cur.right)

            res.append(sum(level_lst) / len(level_lst))

        return res
```

Recursive Solution

```
...
```

[429. N-ary Tree Level Order Traversal]

## Idea: same with level order traversal of binary tree, just need to loop through all the children in corresponding level

Iterative Solution

```
def levelOrder(self, root: 'Node') -> List[List[int]]:
        # N-ary tree, add children in queue one by one 多个for loop去找children其余没区别
        res = []
        if not root: return res
        q = collections.deque([root])
        while q:
            size = len(q)
            level_lst = []
            for _ in range(size):
                cur = q.popleft()
                level_lst.append(cur.val)
                for child in cur.children:
                    q.append(child)
            res.append(level_lst)
        return res
```

Recursive Solution

```
...
```

[515. Find Largest Value in Each Tree Row]

## Idea: Same with 102, just get max value in each level

Iterative Solution

```
def largestValues(self, root: Optional[TreeNode]) -> List[int]:
        # find largest number in each level, store in a list and find max
        res = []
        if not root:
            return res
        q = collections.deque([root])
```

```
        while q:
            level_lst = []
            for _ in range(len(q)):
                cur = q.popleft()
                level_lst.append(cur.val)
                if cur.left: q.append(cur.left)
                if cur.right: q.append(cur.right)
            res.append(max(level_lst))
        return res
```

Recursive Solution

```
...
```

116. Popular Next Right Pointers in Each Node

### Idea: 遇到每层最后一个node之前 先全都连起来 最后一个肯定指向null

Iterative Solution

```
def connect(self, root: 'Optional[Node]') -> 'Optional[Node]':
        if not root: return root
        q = collections.deque([root])
        while q:
            size = len(q)
            for i in range(size):
                cur = q.popleft()
                if cur.left:        q.append(cur.left)
                if cur.right:       q.append(cur.right)
                # when not reaching to the last node, connect all previous nodes
                if i < size - 1:    cur.next = q[0]
            # At last, let the final node point to None
            cur.next = None
        return root
```

Recursive Solution

```
...
```

117. Popular Next Right Pointers in Each Node II

### Idea: No diff with 116

### Idea: No difference with 116.

Iterative Solution

```
def connect(self, root: 'Node') -> 'Node':
        if not root:                return root
        q = collections.deque([root])
        while q:
            size = len(q)
            for i in range(size):
                cur = q.popleft()
                if i < size - 1:    cur.next = q[0]
                if cur.left:        q.append(cur.left)
                if cur.right:       q.append(cur.right)

            cur.next = None
        return root
```

104. Maximum Depth of Binary Tree

### idea: Iterate through nodes in level order, everytime we finished a level, cnt++

Iterative Solution

```
def maxDepth(self, root: Optional[TreeNode]) -> int:
        cnt = 0
        if not root:            return cnt
        # bfs to get the number of levels in the binary tree
        q = collections.deque([root])
        while q:
            size = len(q)
            for i in range(size):
                cur = q.popleft()
                if cur.left:    q.append(cur.left)
                if cur.right:   q.append(cur.right)
            cnt += 1
        return cnt
```

Recursive Solution (1 line)

```
def maxDepth(self, root: Optional[TreeNode]) -> int:
        return 0 if not root else 1 + max(self.maxDepth(root.left),
self.maxDepth(root.right))
        # cnt = 0
        # if not root: return cnt
        # # bfs to get the number of levels in the binary tree
        # cnt = 1 + max(self.maxDepth(root.left), self.maxDepth(root.right))
        # return cnt
```

111. Minimum Depth of Binary Tree

**Idea: 递归要check special cases (一直往左下↙或者一直往右下↘), (still 102变种 记一个level变量更新层数) 用bfs遍历 当遇到第一个node没有左右child的时候就可以返回当前level数了 否则就一直bfs遍历下去**

Iterative Solution

```
def minDepth(self, root: Optional[TreeNode]) -> int:
        level = 0
        if not root:                                return level
        q = collections.deque([root])
        level += 1
        while q:
            size = len(q)
            for i in range(size):
                cur = q.popleft()
                if not cur.left and not cur.right:  return level
                if cur.left:                        q.append(cur.left)
                if cur.right:                       q.append(cur.right)
            level += 1
        return level
```

Recursive Solution

```
def minDepth(self, root: Optional[TreeNode]) -> int:
        if not root:                        return 0
        if not root.left and root.right:    return 1 + self.minDepth(root.right)
        if not root.right and root.left:    return 1 + self.minDepth(root.left)
        return 1 + min(self.minDepth(root.left), self.minDepth(root.right))
```

---

## Invert Binary Tree

226. Invert Binary Tree

### Idea: Iterative is using queue to reverse on every level, Recursive will be start with root and go left and right.

Iterative Solution

```
def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:          return root
        q = collections.deque([root])
        while q:
            for i in range(len(q)):
                cur = q.popleft()
                cur.left, cur.right = cur.right, cur.left
                if cur.left:  q.append(cur.left)
                if cur.right: q.append(cur.right)
        return root
```

Recursive Solution

```
def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if not root:
            return root
        # directly do the swap start with root node
        root.left, root.right = root.right, root.left
        self.invertTree(root.left)
        self.invertTree(root.right)
        return root
```

---

## Symmetric Tree

101. Symmetric Tree

### Idea: check left right with the right left and left left with right right :)

Iterative Solution

```
def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True

        q = collections.deque()
```

```
        q = collections.deque()
        q.append(root.left)
        q.append(root.right)
        while q:
            leftn = q.popleft()
            rightn = q.popleft()
            if not leftn and not rightn:
                # means current is symmetric
                continue
            if not leftn or not rightn or leftn.val != rightn.val:
                # Not symmetric
                return False
            # symmetric adding
            q.append(leftn.left)
            q.append(rightn.right)
            # symmetric adding
            q.append(leftn.right)
            q.append(rightn.left)
        return True
```

Recursive Solution

```
def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True
        # left is None, right, then return False
        # left, right is None, then return False
        # left is None and right is None, return True
        # left and right and left.val != right.val, return False
        # compare left.right with right.left or compare left.left with right.right
        def dfs(left, right):
            if not left and right: return False
            elif left and not right: return False
            elif not left and not right: return True
            elif left.val != right.val: return False
            outside = dfs(left.left, right.right)
            inside = dfs(left.right, right.left)
            return outside and inside
        return dfs(root.left, root.right)
```

222. Count Complete Tree Nodes

**Idea: BFS, update cnt**

### Idea: BFS, update cnt

Iterative Solution

```
def countNodes(self, root: Optional[TreeNode]) -> int:
        # use bfs and a counter to update the number of nodes
        cnt = 0
        if not root:            return cnt
        q = collections.deque([root])
        while q:
            for _ in range(len(q)):
                cur = q.popleft()
                cnt += 1
                if cur.left:    q.append(cur.left)
                if cur.right:   q.append(cur.right)
        return cnt
```

Recursive Solution

### 递归更简单。。跟找max path类似 别忘了count+1 因为root也算进去

```
def countNodes(self, root: Optional[TreeNode]) -> int:
        # use bfs and a counter to update the number of nodes
        cnt = 0
        if not root:            return cnt
        return 1 + self.countNodes(root.left) + self.countNodes(root.right)
```