

BFS使用队列，把每个还没有搜索到的点依次放入队列，然后再弹出队列的头部元素当做当前遍历点。BFS总共有两个模板：如果不需要确定当前遍历到了哪一层，BFS模板如下。

```
1 while queue 不为空:
2     cur = queue.pop()
3     for 节点 in cur的所有相邻节点:
4         if 该节点有效且未访问过:
5             queue.push(该节点)
```

如果要确定当前遍历到了哪一层，BFS模板如下。

这里增加了level表示当前遍历到二叉树中的哪一层了，也可以理解为在一个图中，现在已经走了多少步了。size表示在当前遍历层有多少个元素数，我们把这些元素一次性遍历完，即把当前层的所有元素都向外走了一步。

```
1 level = 0
2 while queue 不为空:
3     size = queue.size()
4     while (size --) {
5         cur = queue.pop()
6         for 节点 in cur的所有相邻节点:
7             if 该节点有效且未被访问过:
8                 queue.push(该节点)
9     }
10    level ++;
```

102 二叉树的层序遍历：

给你一个二叉树，请你返回其按 层序遍历 得到的节点值。（即逐层地，从左到右访问所有节点）。

示例：
二叉树：[3,9,20,null,null,15,7]。

```
3
 / \
9   20
 / \
15  7
```

返回其层序遍历结果：

```
[
  [3],
  [9,20],
  [15,7]
]
```

```
1 def levelOrder(self, root):
2     if not root :
3         return []
4     res = []
5     queue = [root]
6     while queue:
7         # 获取当前队列的长度，这个长度相当于 当前这一层的节点个数
8         size = len(queue)
9         tmp = []
```

```

10         # 将队列中的元素都拿出来(也就是获取这一层的节点), 放到临时list中
11         for _ in xrange(size):
12             r = queue.pop(0)
13             tmp.append(r.val)
14             # 如果节点的左/右子树不为空, 则放入队列中
15             if r.left:
16                 queue.append(r.left)
17             if r.right:
18                 queue.append(r.right)
19         # 将临时list加入最终的结果中
20         res.append(tmp)
21     return res

```

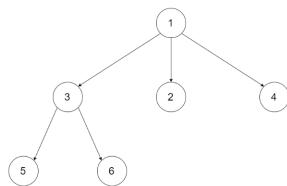
时间复杂度 $O(n)$

空间复杂度是 $O(n)$

429 N叉树的层序遍历, BFS

给定一个 N 叉树, 返回其节点值的层序遍历。(即从左到右, 逐层遍历)。

例如, 给定一个 3 叉树 :



返回其层序遍历:

```

[
  [1],
  [3,2,4],
  [5,6]
]

```

```

1 def levelOrder(self, root):
2     """
3     :type root: Node
4     :rtype: List[List[int]]
5     """
6     if not root:
7         return []
8     res = []
9     queue = [root]
10    while queue:
11        # 获取当前队列的长度, 这个长度相当于 当前这一层的节点个数
12        size = len(queue)
13        tmp = []
14        for i in range(size):
15            cur = queue.pop(0)
16            tmp.append(cur.val)
17            if cur.children:
18                queue.extend(cur.children)
19        res.append(tmp)
20    return res

```

DFS 使用的是递归

```

1 def levelOrder(self, root):

```

```

2      """
3      :type root: TreeNode
4      :rtype: List[List[int]]
5      """
6      if not root :
7          return []
8      res = []
9      def dfs(index, r):
10         # res是[[1],[2,3]], index是3, 就再插入一个空list到res中
11         if len(res) < index :
12             res.append([])
13         # 将当前节点值加入到res中, index是当前层, 假设index是3, 节点值是5,
14         # res是[[1],[2,3],[]], 加入后变成[[1],[2,3],[5]]
15         res[index -1].append(r.val)
16         if r.left:
17             dfs(index+1, r.left)
18         if r.right:
19             dfs(index+1, r.right)
20     dfs(1,root)
21     return res

```

类比 N叉树的层序遍历

```

1  def levelOrder(self, root):
2      """
3      :type root: TreeNode
4      :rtype: List[List[int]]
5      """
6      if not root :
7          return []
8      res = []
9      def dfs(index, r):
10         # res是[[1],[2,3]], index是3, 就再插入一个空list到res中
11         if len(res) < index :
12             res.append([])
13         # 将当前节点值加入到res中, index是当前层, 假设index是3, 节点值是5,
14         # res是[[1],[2,3],[]], 加入后变成[[1],[2,3],[5]]
15         res[index -1].append(r.val)
16         if r.left:
17             dfs(index+1, r.left)
18         if r.right:
19             dfs(index+1, r.right)
20     dfs(1,root)
21     return res

```

时间复杂度: $O(n)$

空间复杂度: $O(h)$. h树的深度