1. # 栈
2. # 栈是一种后进先出的数据结构，我们可以借助list来实现栈
4. stack = [1,2,3,4]
5. stack.append(1) # 入栈
6. stack.pop() # 出栈，返回结果是出栈元素，pop 默认返回最后一个元素
7. peak = stack[-1] # 返回栈顶元素
9. # 队列
10. # 队列是一种先进先出的数据结构，我们可以借助list来实现队列
12. queue = [1,2,3,4]
13. queue.append(1) # 入队
14. queue.pop(0) # 出队，先进先出，返回第一个元素

二叉树是每个结点最多有两个子树的树结构。

一棵深度为k，且有2^k-1个节点的二叉树，称为满二叉树。这种树的特点是每一层上的节点数都是最大节点数。

1. # 声明二叉树节点
2. **class** BiNode(object):
3. **def** \_\_init\_\_(self,element=None,left=None,right=None):
4. self.element = element
5. self.left = left
6. self.right = right
8. # 初始化二叉树
9. **class** BiTree:
10. **def** \_\_init\_\_(self,tree\_node=BiNode):
11. self.root = tree\_node
13. **def** add\_node\_in\_order(self,element):
14. node = BiNode(element)
16. **if** self.root **is** None:
17. self.root = node
18. **else**:
19. node\_queue = list()
20. node\_queue.append(self.root)
21. **while** len(node\_queue):
22. q\_node = node\_queue.pop(0)
23. **if** q\_node.left **is** None:
24. q\_node.left = node
25. **break**
26. **elif** q\_node.right **is** None:
27. q\_node.right = node
28. **break**
29. **else**:
30. node\_queue.append(q\_node.left)
31. node\_queue.append(q\_node.right)

遍历二叉树

前序遍历（递归）

1. **class** Solution:
2. **def** \_\_init\_\_(self):
3. self.ret = []
5. **def** preorderTraversal(self, root):
6. """
7. :type root: TreeNode
8. :rtype: List[int]
9. """
10. **if** root != None:
11. self.ret.append(root.val)
12. self.preorderTraversal(root.left)
13. self.preorderTraversal(root.right)
15. **return** self.ret

中序遍历（递归）

1. **class** Solution:
2. **def** \_\_init\_\_(self):
3. self.ret = []
5. **def** preorderTraversal(self, root):
6. """
7. :type root: TreeNode
8. :rtype: List[int]
9. """
10. **if** root != None:
11. self.preorderTraversal(root.left)
12. self.ret.append(root.val)
13. self.preorderTraversal(root.right)
15. **return** self.ret

后序遍历（递归）

1. **class** Solution:
2. **def** \_\_init\_\_(self):
3. self.ret = []
5. **def** preorderTraversal(self, root):
6. """
7. :type root: TreeNode
8. :rtype: List[int]
9. """
10. **if** root != None:
11. self.preorderTraversal(root.left)
12. self.preorderTraversal(root.right)
13. self.ret.append(root.val)
15. **return** self.ret

前序递归（迭代）

1. **class** Solution:
2. **def** preorderTraversal(self, root):
3. """
4. :type root: TreeNode
5. :rtype: List[int]
6. """
7. result = list()
8. **if** root == None:
9. **return** result
11. stack = list()
12. stack.append(root)
13. **while** len(stack) != 0:
14. top = stack.pop()
15. **if** top.right != None:
16. stack.append(top.right)
17. **if** top.left != None:
18. stack.append(top.left)
19. result.append(top.val)
21. **return** result

中序遍历（迭代）

1. **class** Solution(object):
2. **def** inorderTraversal(self, root):
3. """
4. :type root: TreeNode
5. :rtype: List[int]
6. """
7. ret = []
8. stack = []
9. **while** root **or** stack:
10. **while** root:
11. stack.append(root)
12. root = root.left
13. **if** stack:
14. t = stack.pop()
15. ret.append(t.val)
16. root = t.right
17. **return** ret

后序遍历（迭代）

1. **def** postorderTraversal(self, root):
2. stack = []
3. result = []
4. curr = root
5. **while** stack **or** curr:
6. **if** curr:
7. result.append(curr.val)
8. stack.append(curr.left)
9. curr = curr.right
10. **else**:
11. curr = stack.pop()
12. **return** result[::-1]

翻转二叉树

递归

1. **def** invertTree(self, root: TreeNode) -> TreeNode:
2. **if** **not** root :
3. **return** root
4. **else**:
5. root.left,root.right=self.invertTree(root.right),self.invertTree(root.left)
6. **return** root

翻转二叉树

1. **def** invertTree(self, root: TreeNode) -> TreeNode:
2. **if** root **is** None:
3. **return** root
4. queue = []
5. queue.append(root)
7. **while** queue:
8. next\_queue = []
9. **for** node **in** queue:
10. node.left, node.right = node.right, node.left
11. **if** node.left:
12. next\_queue.append(node.left)
13. **if** node.right:
14. next\_queue.append(node.right)
15. queue = next\_queue
16. **return** root

广度优先，层次遍历

1. **def** levelOrder(self, root):
2. res=[]
3. **if** root==None:
4. **return** res
5. l=[]
6. l.append(root)
7. **while** len(l)>0:
8. temp=[]
9. **for** i **in** range(0,len(l)):
10. node=l[0]
11. **del** l[0]
12. temp.append(node.val)
13. **if** node.left!=None:
14. l.append(node.left)
15. **if** node.right!=None:
16. l.append(node.right)
17. res.append(temp)
18. **return** res

二叉树的最小深度

1. **class** Solution:
2. **def** minDepth(self, root: TreeNode) -> int:
3. **if** **not** root:
4. **return** 0
5. **elif** **not** root.left **and** **not** root.right:
6. **return** 1
7. **elif** **not** root.left **or** **not** root.right:
8. # 左子结点或右子结点存在，不存在的那一边就为0 了，选另一边
9. **return** max(self.minDepth(root.left),self.minDepth(root.right)) +1
10. **else**:
11. # 左右子结点都存在，选更浅的子树
12. **return** min(self.minDepth(root.left),self.minDepth(root.right)) +1

二叉树最大深度

1. **def** maxDepth(self, root):
2. **if** root **is** None:
3. **return** 0
5. ltree = self.maxDepth(root.left)
6. rtree = self.maxDepth(root.right)
8. **return** max(ltree ,rtree )+1

二叉树最近公共祖先

1. **def** lowestCommonAncestor(self, root, p, q):
2. **if** root **is** None **or** root == p **or** root == q:
3. **return** root
4. l = self.lowestCommonAncestor(root.left, p, q)
5. r = self.lowestCommonAncestor(root.right, p, q)
6. **return** root **if** l **and** r **else** l **or** r

二叉搜索树(Binary Search Tree)，又名二叉排序树(Binary Sort Tree)。  
  二叉搜索树是具有有以下性质的二叉树：  
  （1）若左子树不为空，则左子树上所有节点的值均小于或等于它的根节点的值。  
  （2）若右子树不为空，则右子树上所有节点的值均大于或等于它的根节点的值。  
  （3）左、右子树也分别为二叉搜索树。

二叉搜索树最小值

1. **def** findMin(self, root):
2. '''''查找二叉搜索树中最小值点'''
3. **if** root.left:
4. **return** self.findMin(root.left)
5. **else**:
6. **return** root

二叉搜索树的最大值

1. **def** findMax(self, root):
2. '''''查找二叉搜索树中最大值点'''
3. **if** root.right:
4. **return** self.findMax(root.right)
5. **else**:
6. **return** root

二叉搜索树的查找

1. **def** query(self, root, val):
2. '''''二叉搜索树查询操作'''
3. **if** root == None:
4. **return** False
5. **if** root.val == val:
6. **return** True
7. **elif** val < root.val:
8. **return** self.query(root.left, val)
9. **elif** val > root.val:
10. **return** self.query(root.right, val)

二叉搜索树的插入

1. **def** insert(self, root, val):
2. **if** root == None:
3. root = TreeNode(val)
4. **elif** val < root.val:
5. root.left = self.insert(root.left, val)
6. **elif** val > root.val:
7. root.right = self.insert(root.right, val)
8. **return** root

二叉搜索树的最近公共祖先

1. **def** lowestCommonAncestor(self, root, p, q):
2. **if** p.val<root.val **and** q.val<root.val:
3. **return** self.lowestCommonAncestor(root.left,p,q)
4. **elif** p.val>root.val **and** q.val>root.val:
5. **return** self.lowestCommonAncestor(root.right,p,q)
7. **return** root