軟體工程與演算法

使用PYTHOON實作二元收尋樹

* 二元收尋樹的定義

（1）若左子樹不為空，則左子樹上所有節點的值均小於或等於它的根節點的值。  
  （2）若右子樹不為空，則右子樹上所有節點的值均大於或等於它的根節點的值。  
  （3）左、右子樹也分別為二叉搜索樹。

* 以下是使用Pythoon做測試
* class Node(object):
* """節點"""
* def \_\_init\_\_(self, data, left\_child=None, right\_child=None):
* self.data = data
* self.parent = None
* self.left\_child = left\_child
* self.right\_child = right\_child
* class SearchBinaryTree(object):
* """二元樹節點類"""
* def \_\_init\_\_(self):
* self.\_\_root = None
* self.prefix\_branch = '├'
* self.prefix\_trunk = '|'
* self.prefix\_leaf = '└'
* self.prefix\_empty = ''
* self.prefix\_left = '─L─'
* self.prefix\_right = '─R─'
* def is\_empty(self):
* return not self.\_\_root
* @property
* def root(self):
* return self.\_\_root
* @root.setter
* def root(self, value):
* self.\_\_root = value if isinstance(value, Node) else Node(value)
* def show\_tree(self):
* if self.is\_empty():
* print('空的二元樹')
* return
* print('-' \* 20)
* print(self.\_\_root.data)
* self.\_\_print\_tree(self.\_\_root)
* print('-' \* 20)
* def insert(self, root, value):
* """二叉搜索树插入节点-递归"""
* node = value if isinstance(value, Node) else Node(value)
* if self.is\_empty():
* self.root = node
* return
* if root is None:
* root = node
* elif node.data < root.data:
* root.left\_child = self.insert(root.left\_child, value)
* root.left\_child.parent = root
* elif node.data > root.data:
* root.right\_child = self.insert(root.right\_child, value)
* root.right\_child.parent = root
* return root
* def insert\_normal(self, value):
* """二叉搜索树插入节点-非递归"""
* node = value if isinstance(value, Node) else Node(value)
* if self.is\_empty():
* self.root = node
* return
* else:
* current\_node = self.root
* while True:
* if node.data < current\_node.data:
* if current\_node.left\_child:
* current\_node = current\_node.left\_child
* else:
* current\_node.left\_child = node
* node.parent = current\_node
* break
* elif node.data > current\_node.data:
* if current\_node.right\_child:
* current\_node = current\_node.right\_child
* else:
* current\_node.right\_child = node
* node.parent = current\_node
* break
* else:
* break
* def search(self, root, data):
* """二叉搜索树的查询操作"""
* if root is None:
* return False
* if root.data == data:
* return True
* elif data < root.data:
* return self.search(root.left\_child, data)
* elif data > root.data:
* return self.search(root.right\_child, data)
* def get\_max(self, root):
* """查找二叉搜索树的最大值"""
* if self.is\_empty():
* return
* return self.get\_max(root.right\_child) if root.right\_child else root.data
* def get\_min(self, root):
* """查找二叉搜索树的最小值"""
* if self.is\_empty():
* return
* return self.get\_min(root.left\_child) if root.left\_child else root.data
* def levelorder\_traversal(self):
* """层序遍历"""
* if self.is\_empty():
* return
* queue = list()
* queue.insert(0, self.\_\_root)
* while len(queue):
* cur = queue.pop()
* print(cur.data, end=' ')
* if cur.left\_child is not None:
* queue.insert(0, cur.left\_child)
* if cur.right\_child is not None:
* queue.insert(0, cur.right\_child)
* print()
* def preorder\_traversal(self, node):
* """先序遍历"""
* if node is None:
* return
* print(node.data, end=' ')
* self.preorder\_traversal(node.left\_child)
* self.preorder\_traversal(node.right\_child)
* def inorder\_traversal(self, node):
* """中序遍历"""
* if node is None:
* return
* self.inorder\_traversal(node.left\_child)
* print(node.data, end=' ')
* self.inorder\_traversal(node.right\_child)
* def postorder\_traversal(self, node):
* """后序遍历"""
* if node is None:
* return
* self.postorder\_traversal(node.left\_child)
* self.postorder\_traversal(node.right\_child)
* print(node.data, end=' ')
* def \_\_print\_tree(self, node, prefix=None):
* if prefix is None:
* prefix, prefix\_left\_child = '', ''
* else:
* prefix = prefix.replace(self.prefix\_branch, self.prefix\_trunk)
* prefix = prefix.replace(self.prefix\_leaf, self.prefix\_empty)
* prefix\_left\_child = prefix.replace(self.prefix\_leaf, self.prefix\_empty)
* if self.has\_child(node):
* if node.right\_child is not None:
* print(prefix + self.prefix\_branch + self.prefix\_right + str(node.right\_child.data))
* if self.has\_child(node.right\_child):
* self.\_\_print\_tree(node.right\_child, prefix + self.prefix\_branch + ' ')
* else:
* print(prefix + self.prefix\_branch + self.prefix\_right)
* if node.left\_child is not None:
* print(prefix + self.prefix\_leaf + self.prefix\_left + str(node.left\_child.data))
* if self.has\_child(node.left\_child):
* prefix\_left\_child += ' '
* self.\_\_print\_tree(node.left\_child, self.prefix\_leaf + prefix\_left\_child)
* else:
* print(prefix + self.prefix\_leaf + self.prefix\_left)
* def has\_child(self, node):
* return node.left\_child is not None or node.right\_child is not None
* def \_\_str\_\_(self):
* return str(self.\_\_class\_\_)
* if \_\_name\_\_ == '\_\_main\_\_':
* tree = SearchBinaryTree()
* data = [50, 77, 55, 29, 10, 50, 30, 66, 18, 80, 51, 18, 90]
* for i in data:
* *# tree.insert(tree.root, i)*
* tree.insert\_normal(i)
* tree.show\_tree()
* print('层次遍历： ', end='')
* tree.levelorder\_traversal()
* print('先序遍历： ', end='')
* tree.preorder\_traversal(tree.root)
* print()
* print('中序遍历： ', end='')
* tree.inorder\_traversal(tree.root)
* print()
* print('后序遍历： ', end='')
* tree.postorder\_traversal(tree.root)
* print()
* print("查询结果为：", tree.search(tree.root, 50))
* print("查询结果为：", tree.search(tree.root, 500))
* print("二叉搜素树的最大值是：", tree.get\_max(tree.root))
* print("二叉搜素树的最小值是：", tree.get\_min(tree.root))