

Федеральное государственное автономное образовательное учреждение
высшего образования «Национальный исследовательский университет
ИТМО»

Факультет инфокоммуникационных технологий

ОТЧЕТ

по лабораторной работе №2. «Двоичные деревья поиска» по
дисциплине «Алгоритмы и структуры данных»

Выполнил студент 1 курса, группы К3140

Байков Иван

Преподаватель: Харьковская Татьяна Александровна

05 Июня, 2022 года, г. Санкт-Петербург

Задание 1

Описание

В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (in-order), прямой (pre-order) и обратный (post-order).

Исходный код

```
class BinaryTreeNode:
    def __init__(self, data):
        self.data = data
        self.left = None
        self.right = None

class BinaryTree:
    def __init__(self, root_data):
        self.root = BinaryTreeNode(root_data)

    def __str__(self):
        preorder_recursive = str(self.preorder_dfs_recursive(self.root))
        inorder_recursive = str(self.inorder_dfs_recursive(self.root))
        postorder_recursive = str(self.postorder_dfs_recursive(self.root))

        return f"""
        \nPre-Order\n{preorder_recursive}
        \nIn-Order\n{inorder_recursive}
        \nPost-Order\n{postorder_recursive}
        """

    def preorder_dfs_recursive(self, start_node, result=[]):
        if start_node is not None:
            result.append(start_node.data)
            self.preorder_dfs_recursive(start_node.left, result)
            self.preorder_dfs_recursive(start_node.right, result)
        return result

    def inorder_dfs_recursive(self, start_node, result=[]):
        if start_node is not None:
```

```

def postorder_dfs_recursive(self, start_node, result=[]):
    if start_node is not None:
        self.postorder_dfs_recursive(start_node.left, result)
        self.postorder_dfs_recursive(start_node.right, result)
        result.append(start_node.data)
    return result

#           2
#         /  \
#        1    3

binary_tree = BinaryTree(2)

binary_tree.root.left = BinaryTreeNode(1)
binary_tree.root.right = BinaryTreeNode(3)

print(binary_tree)

```

Решение

Мы обходим дерево по определенным порядкам, стартовой точкой (entry point) всегда является корень, но он не обязан быть первым в списке обхода. Обход реализован 3 строчками кода, где одна – это добавление результата, другие две – рекурсивный вызов левого или правого поддерева. В зависимости от типа обхода эти функции выполняются в разном порядке.

Тесты

```
Pre-Order
```

```
[2, 1, 3]
```

```
In-Order
```

```
[1, 2, 3]
```

```
Post-Order
```

```
[1, 3, 2]
```

Взял простое дерево [2,1,3], чтобы легче показать на примере.

Вывод

Было полезно попрактиковаться в реализации алгоритмов обхода, чтобы лучше понимать такую полезную структуру данных и как с ней взаимодействовать.

Задание 6-7

Описание

В данной задаче необходимо проверить, правильно ли реализована структура BST. В 7 задаче усложнение – деревья могут содержать равные ключи.

Исходный код

```
1  from typing import List
2  import sys
3
4
5  class TreeNode():
6      root: int
7      children: List[int]
8
9      def __init__(self, root, left, right):
10         self.root = root
11         self.children = [left, right]
12
13
14     tree = []
15     _input = ""
16     2 1 2
17     1 -1 -1
18     3 -1 -1
19
20     _input = _input.split("\n")
21     n = int(_input[0])
22
23
24     def is_valid(tree, node, min_root, max_root):
25         if node == -1:
26             return True
27         if tree[node].root <= min_root or tree[node].root >= max_root:
28             return False
29
30         result_left = is_valid(tree, tree[node].children[0], min_root, tree[node].root)
31         result_right = is_valid(tree, tree[node].children[1], tree[node].root, max_root)
```

```

    result_left = is_valid(tree, tree[node].children[0], min_root, tree[node].root)
    result_right = is_valid(tree, tree[node].children[1], tree[node].root, max_root)

    return result_left and result_right

for line in _input[1:]:
    root, left, right = list(map(int, line.split(" ")))
    tree.append(TreeNode(root, left, right))

if n == 0:
    print("CORRECT")
else:
    if is_valid(tree, 0, -sys.maxsize - 1, sys.maxsize + 1):
        print("CORRECT")
    else:
        print("INCORRECT")

```

Решение

При выполнении In-order DFS каждый текущий узел во время обхода можно добавлять в массив, тогда в конце обхода можно наглядно убедиться, является ли данное дерево двоичным деревом поиска. Суть в том, что при таком обходе, каждый следующий элемент массива будет больше или равен (7) предыдущему, поэтому простым циклом можно это проверить.

Тесты

```

tree = []
_input = """3
2 1 2
1 -1 -1
3 -1 -1"""

```

CORRECT

```

tree = []
_input = """3
2 1 2
1 -1 -1
2 -1 -1"""

```

INCORRECT

Были проверены два теста из условия, один на CORRECT, другой на INCORRECT.

Вывод

Не самая сложная задача, если решать через рекурсивный подход. Помогает детальнее изучить BST и узнать некоторые свойства. Асимптотика алгоритма – $O(n)$, где n – количество узлов.

Задание 11-15

Описание

Если объединить все задания, то нужно реализовать AVL дерево, в котором можно добавлять/удалять узлы, искать высоту, след/пред узел, делать левый и правый повороты и т.д.

Исходный код

На гитхабе, некоторый функционал в решении.

Решение

```
def insert(self, data):
    new_node = Node(data)

    if self.root is not None:
        if data < self.root.data:
            self.root.left.insert(data)
        elif data > self.root.data:
            self.root.right.insert(data)
        else:
            print("Node is already in AVL")
            return
    else:
        self.root = new_node
        self.root.left = AVLTree()
        self.root.right = AVLTree()

    self.rebalance()
```

В первую очередь реализована вставка в дерево с автоматической ребалансировкой, выполняется с помощью массива:


```
for i in [8, 4, 2, 6, 3, 5, 1, 7, 9, 1, 0, 11, 2, 13]:  
    tree.insert(i)
```

```
def delete(self, data):  
  
    if self.root is not None:  
        if self.root.data == data:  
            print("Deleting node: ", data)  
  
            if self.root.left.root is None and self.root.right.root is None:  
                self.root = None  
            elif self.root.left.root is None and self.root.right.root is not None:  
                self.root = self.root.right.root  
            elif self.root.left.root is not None and self.root.right.root is None:  
                self.root = self.root.left.root  
            else:  
                replacement = self.next(self.root)  
                if replacement is not None:  
                    self.root.data = replacement.data  
                    self.root.right.delete(replacement.data)  
  
                self.rebalance()  
                return  
        elif data < self.root.data:  
            self.root.left.delete(data)  
        elif data > self.root.data:  
            self.root.right.delete(data)  
  
        self.rebalance()  
    else:  
        return
```

Для удаления сначала рекурсивно проходимся по дереву и ищем узел с нужным значением, затем заменяем узел на None либо заменяем (если есть дочерние узлы) его данные на дочерние. Для поиска дочерних используем метод next().

```

def next(self, root):
    print('asd', root)

    root = root.right.root

    if root is not None:
        while root.left is not None:
            if root.left.root is None:
                return root
            else:
                root = root.left.root

    return root.data

```

Для балансировки делаем левый или правый повороты.

```

def rotate_left(self):
    print("Rotating ", self.root.data, " left")

    a = self.root
    b = self.root.right.root
    c = b.left.root

    self.root = b
    b.left.root = a
    a.right.root = c

```

Тесты

```
if __name__ == "__main__":  
    tree = AVLTree()  
  
    print("Insert")  
    for i in [8, 4, 2, 6, 3, 5, 1, 7, 9, 1, 0, 11, 2, 13]:  
        tree.insert(i)  
  
    print(tree.in_order())  
  
    print("Delete")  
    tree.delete(3)  
    tree.delete(4)  
    tree.delete(5)  
    print(tree.in_order())  
    print(tree.root.right.height)
```

```
Insert
Rotating 8 right
Rotating 8 right
Node is already in AVL
Rotating 6 left
Node is already in AVL
Rotating 9 left
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13] ✓
Delete
Deleting node: 3
Rotating 2 right
Deleting node: 4
Deleting node: 5
Deleting node: 5
Deleting node: 6
[0, 1, 2, 6, 7, 8, 9, 11, 13] ✓
2
```

Взяты случайные небольшие значения для лучшего понимания происходящего. Галочками отмечены состояния дерева во время тестов. Как видно они балансируются до BST.

Вывод

Проработан основной функционал AVL деревьев, получены практические знания о реализации и удобстве хранения данных в таком виде.

Задание 17

Описание

В этой задаче ваша цель – реализовать структуру данных для хранения набора целых чисел и быстрого вычисления суммы элементов в заданном диапазоне.

Методы: add, del, find, sum.

Исходный код

Так же на гитхабе.

Решение

Каждая вершина имеет ключ, представляющий целое число, сумму, равную сумме всех ключей в поддереве, указатель на его левую дочернюю вершину, правый указатель на его правую дочернюю вершину и родительский элемент, указывающий на родительскую вершину. Для удобства дерево делится на 2 поддерева используя указанный ключ.

Тесты

```
6
+ 3
+ 10
- 8
? 8
Not found
+ 3
s 1 9
3
```

```
5
+ 491572259
? 491572259
? 899375874
s 310971296 877523306
+ 352411209 Found
Not found
491572259
```

Тесты из условия + рандомные, проверяют весь функционал.

Вывод

Реализовано Splay дерево, позволяющее хранить и удаления данных с возможностью быстрого вычисления суммы элементов в заданном диапазоне. Непростое задание с основной сложность в виде реализации Splay дерева с ключами, split и merge методами.

.