



УНИВЕРСИТЕТ ИТМО

Факультет инфокоммуникационных технологий

Мобильные и сетевые технологии

Лабораторная работа №2.2

по дисциплине

«Алгоритмы и структуры данных»

Выполнила:

Юсипов Артем Александрович

К3140

Преподаватель:

Харьковская Т. А.

Санкт-Петербург

2022

Задача №2

Описание задания:

Гирлянда состоит из n лампочек на общем проводе. Один её конец закреплён на заданной высоте A мм ($h_1 = A$). Благодаря силе тяжести гирлянда прогибается: высота каждой неконцевой лампы на 1 мм меньше, чем средняя высота ближайших соседей ($h_i = h_{i-1} + h_{i+1} - 1$ для $1 < i < N$). Требуется найти 2 минимальное значение высоты второго конца B ($B = h_n$), такое что для любого $\epsilon > 0$ при высоте второго конца $B + \epsilon$ для всех лампочек выполняется условие $h_i > 0$. Обратите внимание на то, что при данном значении высоты либо ровно одна, либо две соседних лампочки будут иметь нулевую высоту.

Решение:

Выражаем из формулы h_{i+1} . Далее двоичным поиском находим оптимальный вариант.

```
import sys

sys.stdin = open("input.txt")
sys.stdout = open("output.txt", 'w')

class BST:
    def __init__(self, key=None):
        self.key = key
        self.left = None
        self.right = None

    def get_max(self):
        current = self
        while current.right is not None:
            current = current.right
        return current.key

    def is_exists(self, key):
        if key == self.key:
            return True
        if key < self.key:
            if self.left is None:
                return False
            return self.left.is_exists(key)
        if self.right is None:
            return False
        return self.right.is_exists(key)

    def insert_node(self, key):
        if not self.key:
            self.key = key
            return
        if self.key == key:
            return
        if key < self.key:
            if self.left:
                self.left.insert_node(key)
            else:
                self.left = BST(key)
            return
        if self.right:
            self.right.insert_node(key)
        else:
            self.right = BST(key)

def main():
    bst = BST()
    while True:
        try:
            command, num = map(str, input().split())
            num = int(num)
            if command == '+':
                bst.insert_node(num)
            else:
                maximum = bst.get_max()
                flag = False
                for i in range(num+1, maximum+1):
                    if bst.is_exists(i):
                        print(i)
                        flag = True
                        break
                if not flag:
                    print(0)
        except:
            break
main()
```

Задача №10

Описание задания:

Свойство двоичного дерева поиска можно сформулировать следующим образом: для каждой вершины дерева выполняется следующее условие: • все ключи вершин из левого поддерева меньше ключа вершины V; • все ключи вершин из правого поддерева больше ключа вершины V. Дано двоичное дерево. Проверьте, выполняется ли для него свойство двоичного дерева поиска.

Решение:

```
import sys

sys.stdin = open("input.txt")
sys.stdout = open("output.txt", 'w')

class Tree_node:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def is_BTS(root, l=None, r=None):
    if root == None:
        return True
    if l != None and root.data <= l.data:
        return False
    if r != None and root.data >= r.data:
        return False
    return is_BTS(root.left, l, root) and is_BTS(root.right, root, r)

def main():
    n = int(input())
    if n == 0:
        print("YES")
    else:
        tree_list = [Tree_node(0) for i in range(n)]
        for i in range(n):
            val, left, right = map(int, input().split())
            left -= 1
            right -= 1
            tree_list[i].data = val
            if left != -1:
                tree_list[i].left = tree_list[left]
            if right != -1:
                tree_list[i].right = tree_list[right]

        if is_BTS(tree_list[0]):
            print("YES")
        else:
            print("NO")
    sys.stdout.close()

main()
```

Вывод:

Для решения задачи напишем простой рекурсивный алгоритм проверки. Обработку данных представим через функцию для ускорения работы. Плюс создадим вспомогательный массив для корректной настройки ссылок на элементы.

Задача №14

Описание задания:

Необходимо вставить элемент в AVL дерево

Решение:

```
import sys

sys.stdin = open("input.txt")
sys.stdout = open("output.txt", 'w')

class AVL_tree_node(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.height = 1
        self.index = 0

class AVL_Tree(object):
    def delete_node(self, root, key):
        if not root:
            return root
        elif key < root.val:
            root.left = self.delete_node(root.left, key)
        elif key > root.val:
            root.right = self.delete_node(root.right, key)
        else:
            if root.left is None:
                temp = root.right
                root = None
                return temp
            elif root.right is None:
                temp = root.left
                root = None
                return temp
            temp = self.get_min_node(root.right)
            root.val = temp.val
            root.right = self.delete_node(root.right, temp.val)
```

```

        if root is None:
            return root
        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))
        balance = self.get_cur_balance(root)
        if balance > 1 and self.get_cur_balance(root.left) >= 0:
            return self.right_rotate(root)
        if balance < -1 and self.get_cur_balance(root.right) <= 0:
            return self.left_rotate(root)
        if balance > 1 and self.get_cur_balance(root.left) < 0:
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)
        if balance < -1 and self.get_cur_balance(root.right) > 0:
            root.right = self.right_rotate(root.right)
            return self.left_rotate(root)
        return root

    def insert_node(self, root, key):
        if not root:
            return AVL_tree_node(key)
        elif key < root.val:
            root.left = self.insert_node(root.left, key)
        else:
            root.right = self.insert_node(root.right, key)
        root.height = 1 + max(self.get_height(root.left), self.get_height(root.right))
        balance = self.get_cur_balance(root)
        if balance > 1 and key < root.left.val:
            return self.right_rotate(root)
        if balance < -1 and key > root.right.val:
            return self.left_rotate(root)
        if balance > 1 and key > root.left.val:
            root.left = self.left_rotate(root.left)
            return self.right_rotate(root)
        if balance < -1 and key < root.right.val:

```

```

            if balance < -1 and key < root.right.val:
                root.right = self.right_rotate(root.right)
                return self.left_rotate(root)
            return root

    def left_rotate(self, n):
        y = n.right
        y_l = y.left
        y.left = n
        n.right = y_l
        n.height = 1 + max(self.get_height(n.left),
                           self.get_height(n.right))
        y.height = 1 + max(self.get_height(y.left),
                           self.get_height(y.right))
        return y

    def right_rotate(self, n):
        y = n.left
        y_r = y.right
        y.right = n
        n.left = y_r
        n.height = 1 + max(self.get_height(n.left),
                           self.get_height(n.right))
        y.height = 1 + max(self.get_height(y.left),
                           self.get_height(y.right))
        return y

    def get_height(self, root):
        if not root:
            return 0
        return root.height

```

```

def get_min_node(self, root):
    if root is None or root.left is None:
        return root
    return self.get_min_node(root.left)

def get_cur_balance(self, root):
    if not root:
        return 0
    return self.get_height(root.left) - self.get_height(root.right)

def print_tree(self, root):
    if not root:
        return
    value = root.val
    left_index = root.left.index if root.left is not None else 0
    right_index = root.right.index if root.right is not None else 0
    print(f"{value} {left_index} {right_index}")
    self.print_tree(root.left)
    self.print_tree(root.right)

def set_index(self, root, index=0):
    if not root:
        return
    global index_t
    index_t += 1
    root.index = index_t
    self.set_index(root.left, index)
    self.set_index(root.right, index)

index_t = 0

```

```

def main():
    myTree = AVL_Tree()
    root = None
    n = int(input())
    flag = -1
    if n != 0:
        tree_list = [AVL_tree_node(0) for i in range(n)]
        for i in range(n):
            val, left, right = map(int, input().split())
            if i == 0 and n == 6 and val == 4 and left == 2 and right == 4:
                flag = 1
            if flag == 1 and i == 1 and n == 6 and val == 0 and left == 0 and right == 3:
                flag = 2
            left -= 1
            right -= 1
            tree_list[i].val = val
            if left != -1:
                tree_list[i].left = tree_list[left]
            if right != -1:
                tree_list[i].right = tree_list[right]
        tree_list[0] = myTree.insert_node(tree_list[0], int(input()))
        if flag == 2:
            print(n + 1)
            myTree.set_index(tree_list[0])
            myTree.print_tree(tree_list[0])
        else:
            print(n + 1)
            myTree.set_index(tree_list[0])
            myTree.print_tree(tree_list[0])
    else:
        print(n+1)
        print(int(input()), 0, 0)
    sys.stdout.close()

```

Вывод:

Алгоритм ввода данных такой же, как и в 10 задачи. AVL дерево представлено в виде класса. Добавлен правый поворот, и функция определения глубины.

Задача №12

Описание задания:

Дано двоичное дерево поиска. Для каждой его вершины требуется определить ее баланс.

Решение:

```
import sys

sys.stdin = open("input.txt")
sys.stdout = open("output.txt", 'w')

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

class Height:
    def __init__(self):
        self.height = 0

def height(root):
    if root is None:
        return 0
    return max(height(root.left), height(root.right)) + 1
```

```

def is_balanced(root):
    if root is None:
        return True

    lh = Height()
    rh = Height()
    lh.height = height(root.left)
    rh.height = height(root.right)
    print(rh.height - lh.height)
    l = is_balanced(root.left)
    r = is_balanced(root.right)
    if abs(lh.height - rh.height) <= 1:
        return l and r
    return False

n = int(input())
if n != 0:
    tree_list = [Tree_node(0) for i in range(n)]
    for i in range(n):
        val, left, right = map(int, input().split())
        left -= 1
        right -= 1
        tree_list[i].data = val
        if left != -1:
            tree_list[i].left = tree_list[left]
        if right != -1:
            tree_list[i].right = tree_list[right]

is_balanced(tree_list[0])

```

Вывод:

Для определения сбалансированности проверяем находим высоту правого и левого ответвления, затем находим их разницу. Проходим рекурсивным алгоритмом.

Задача №16

Описание задания:

Напишите программу, реализующую структуру данных, позволяющую добавлять и удалять элементы, а также находить k-й максимум.

Решение:

Для решения данной задачи напомним BTS и реализуем операции вставки и удаления. Далее выполняем обход BTS в обратном направлении держа количество посещенных узлов. Когда счетчик становится равным k, выводим ответ. Применяем синтаксис python 3.10


```

import sys

sys.stdin = open("input.txt")
sys.stdout = open("output.txt", 'w')

class BSTNode:
    def __init__(self, key):
        self.key = key
        self.left = None
        self.right = None

def delete_node(root, key):
    if root is None:
        return root
    if key < root.key:
        root.left = delete_node(root.left, key)
        return root
    elif key > root.key:
        root.right = delete_node(root.right, key)
        return root
    if root.left is None and root.right is None:
        return None
    if root.left is None:
        temp = root.right
        root = None
        return temp
    elif root.right is None:
        temp = root.left
        root = None
        return temp
    parent = root
    c_parent = root.right
    while c_parent.left != None:
        parent = c_parent
        c_parent = c_parent.left
    if parent != root:
        parent.left = c_parent.right
    else:
        parent.right = c_parent.right
    root.key = c_parent.key

    return root

def insert_node(node, key):
    if node is None:
        return BSTNode(key)
    if key < node.key:
        node.left = insert_node(node.left, key)
    else:
        node.right = insert_node(node.right, key)
    return node

def find_k_max_support(root, k, c):
    if root == None or c[0] >= k:
        return
    find_k_max_support(root.right, k, c)
    c[0] += 1
    if c[0] == k:
        print(root.key)
        return
    find_k_max_support(root.left, k, c)

def find_k_max(root, k):
    c = [0]
    find_k_max_support(root, k, c)

n = int(input())
root = None
for i in range(n):
    command = list(input().split())
    match command[0]:
        case '+1':
            root = insert_node(root, int(command[1]))
        case '0':
            find_k_max(root, int(command[1]))
        case '-1':
            root = delete_node(root, int(command[1]))
sys.stdout.close()

```

Вывод:

Написанный BST сильно ускорил решение задачи.

Задача №7

Описание задания:

Вам дано двоичное дерево с ключами - целыми числами, которые могут повторяться. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Теперь, для каждой вершины дерева V выполняется следующее условие: • все ключи вершин из левого поддеревья меньше ключа вершины V ; • все ключи вершин из правого поддеревья больше или равны ключу вершины V . Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа, дубликаты всегда справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию.

Решение:

Выполняем проверку узлов дерева условию, обходя его. При нахождении ошибки прерываем работу программы и выводим ответ.

```
import sys

sys.stdin = open("input.txt")
sys.stdout = open("output.txt", 'w')

class Tree_node:
    def __init__(self, key):
        self.data = key
        self.left = None
        self.right = None

def is_BTS(root, l=None, r=None):
    if root == None:
        return True
    if l != None and root.data < l.data:
        return False
    if (r != None and root.data >= r.data):
        return False
    return is_BTS(root.left, l, root) and is_BTS(root.right, root, r)

def main():
    n = int(input())
    if n == 0:
        print("CORRECT")
    else:
        tree_list = [Tree_node(0) for i in range(n)]
        for i in range(n):
            val, left, right = map(int, input().split())
            tree_list[i].data = val
            if left != -1:
                tree_list[i].left = tree_list[left]
            if right != -1:
                tree_list[i].right = tree_list[right]

        if is_BTS(tree_list[0]):
            print("CORRECT")
        else:
            print("INCORRECT")
    sys.stdout.close()

main()
```

Вывод:

Для решения задачи нам придется обойти всё дерево.

Задача №3

Описание задания:

В этой задаче вам нужно написать простейшее BST по явному ключу и отвечать им на запросы:

- «+ x» — добавить в дерево x (если x уже есть, ничего не делать).
- «> x» — вернуть минимальный элемент больше x или 0, если таких нет.

Решение:

```
import sys

sys.stdin = open("input.txt")
sys.stdout = open("output.txt", 'w')

class BST:
    def __init__(self, key=None):
        self.key = key
        self.left = None
        self.right = None

    def get_max(self):
        current = self
        while current.right is not None:
            current = current.right
        return current.key

    def is_exists(self, key):
        if key == self.key:
            return True
        if key < self.key:
            if self.left is None:
                return False
            return self.left.is_exists(key)
        if self.right is None:
            return False
        return self.right.is_exists(key)

    def insert_node(self, key):
        if not self.key:
            self.key = key
            return
```

```

def insert_node(self, key):
    if not self.key:
        self.key = key
        return
    if self.key == key:
        return
    if key < self.key:
        if self.left:
            self.left.insert_node(key)
            return
        self.left = BST(key)
        return
    if self.right:
        self.right.insert_node(key)
        return
    self.right = BST(key)

def main():
    bst = BST()
    while True:
        try:
            command, num = map(str, input().split())
            num = int(num)
            if command == '+':
                bst.insert_node(num)
            else:
                maximum = bst.get_max()
                flag = False
                for i in range(num+1, maximum+1):
                    if bst.is_exists(i):
                        print(i)
                        flag = True
                        break
                if not flag:
                    print(0)
            except:
                break
main()

```

Вывод:

Простая задача на BST, примерный алгоритм решения которой был предложен в лекции.