Министерство образования и науки Российской Федерации
Федеральное государственное автономное образовательное учреждение высшего образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Мегафакультет трансляционных информационных технологий
Факультет инфокоммуникационных технологий

Дисциплина: Алгоритмы и структуры данных

Отчет по Лабораторной работе №2

Выполнила: Микулина Алиса Романовна

Группа: К3143, 1 курс

Преподаватель: Харьковская Татьяна Александровна

Санкт-Петербург

23.06.2022

Описание задания

Задание 3.

BST по явному ключу (насколько я поняла, все что с ключом, это декартач). Уметь добавлять элементы и возвращать минимальный элемент больше x.

Задание 4.

По заданию вроде как неявный декартач, но такой функционал как тут требуется чет не очень понятно как реализовывать в неявном, поэтому я допилила явный. Задача уметь добавлять элементы и возвращать катый по возрастанию элемент.

Задание 5.

Адская задача с адским количеством запросов. Бинарное дерево поиска, добавляем, удаляем вершинка, проверяем наличие, выводим следующий и предыдущий данному элементы.

Задание 6.

Имеем какое-то дерево, претендующее на звание двоичного дерева поиска. Считываем, строим, проверяем, двоична це штука али нет.

Задание 7.

Задачка-мем на основе предыдущей. Тут одно отличие от обычного BST – в правом поддереве могут лежать ноды больше или равные вершинке.

Задание 8.

Имеем двоичное дерево поиска, нужно посчитать глубину. Реализовано на основе BST из пятой задачи.

Задание 9.

Удаление поддеревьев. Я его тоже реализовала на BST, хотя и на декартовом было бы удобно. После остальных задач что-то слишком просто.

Задание 16.

Задачка на нахождение катого максимума. Сейчас уже поняла, что декартовым деревом быстрее, удобнее и в 200 раз проще, но уже нет ни сил ни времени переписывать.

Описание решения и исходный код

Задача 3.

Как ни странно, эта задача была решена предпоследней, потому что вообще было страшно приближаться к чему-либо связанному с декартовым деревом. На деле оказалось все не так страшно, а наоборот, проще чем обычное дерево раз в 10000000000000000000…

Очень долго въезжала в суть двух координат, потом кааааак поняла, аж приятно стало. С нуля, конечно, до такого кода не додумаешься, но когда читаешь и слушаешь как это делается, через пару часов становится получше.

Ну тут на самом деле ничего интересного. Split, Merge и поиск следующего числа с помощью merge. У меня был просто непередаваемый шок, когда я узнала, что можно разделить по данному элементу нашу “дучку”, и наименьшее значение правого полученного дерева будет искомым.

Долго не понимала почему дерево нормально не строится. Оказалось, что у меня find и next просто разрезали сплитом дерево и не склеивали обратно. Такая мелкая ошибка, а я из-за нее сегодня ночью плакала…

В общем оно работает, я довольна 😊

```
from random import randint

class Node:
    def __init__(self):
        self.data = None
        self.priority = randint(1, 2 ** 64)
        self.left = None
        self.right = None
        self.size = 1




class Treap:
    def __init__(self):
        self.root = None


    def insert(self, elem):
        new_node = Node()
        new_node.data = elem
        if self.root == None:
            self.root = new_node
```

```
            return
        if self.if_exists(elem) == True:
            return
        left, right = self.split(self.root, elem - 1)
        self.root = self.merge(left, new_node)
        self.root = self.merge(self.root, right)
        return


    def deepart(self, rut): # it's called deepart because it looks like it will
depart me to hell yet it's art
        left = 0
        right = 0
        if rut.left != None:
            left = rut.left.size
        if rut.right != None:
            right = rut.right.size
        return left + right + 1


    def if_exists(self, elem):
        left, right = self.split(self.root, elem - 1)
        if right == None:
            return False
        r = right
        while right.left != None:
            right = right.left
        if right.data != elem:
            self.root = self.merge(left, r)
            return False
        else:
            self.root = self.merge(left, r)
            return True


    def merge(self, left, right):
        if left == None:
            return right
        elif right == None:
            return left
        elif left.priority > right.priority:
            left.right = self.merge(left.right, right)
            ans = left
        else:
            right.left = self.merge(left, right.left)
            ans = right
        ans.size = self.deepart(ans)
        return ans


    def split(self, rut, elem):
```

```
        if rut == None:
            return (None, None)
        if elem < rut.data:
            left, rut.left = self.split(rut.left, elem)
            right = rut
            right.size = self.deepart(right)
            return (left, right)
        else:
            rut.right, right = self.split(rut.right, elem)
            left = rut
            left.size = self.deepart(left)
            return (left, right)


    def next_x(self, elem): # WHY ON EARTH IT IS 7 LINES LONG NOT 30
        left, right = self.split(self.root, elem)
        if right == None:
            return 0
        r = right
        while right.left != None:
            right = right.left
        self.root = self.merge(left, r)
        return right.data
```

Задача 4.

В задании написано: BST по неявному ключу. Я честное слово искала информацию два дня, но так и не поняла, как на декартаче с неявным ключом можно хранить элементы в порядке возрастания значения, а не x.

Поэтому, как бы ни было грустно, просто делаем новую функцию в дуче из третьего задания. Но чтобы эта функция работала, нам нужно знать размер каждого поддерева каждого элемента. А это сложно потому что дерево постоянно разрезается, совмещается итд итп… В итоге тыкаясь во все части кода я нашла таки в какие моменты нужно пересчитывать глубину и оно даже стало работать…

По факту алгоритм нахождения катого элемента достаточно простой, просто смотрим на наше k, на глубину левого поддерева, и если k больше глубины, то из k вычитается глубина левого поддерева и вызывается заново эта же функция, но уже от правого поддерева. Иначе вызываем от левого ничего не уменьшая. Когда k становится равен глубине левого поддерева, то ура, мы нашли вершинку!!!

Выводим, и идем спать, потому что это для меня было последнее задание в лабе.

```
    def give_kth(self, k, rut='aa'):
        if rut == 'aa':
            rut = self.root
        if rut.left != None:
            size_l = rut.left.size
        else:
            size_l = 0
        if size_l == k:
            return rut.data
        elif size_l > k:
            rut = rut.left
            return self.give_kth(k, rut)
        elif size_l < k:
            rut = rut.right
            k = k - size_l - 1
            return self.give_kth(k, rut)
```

Задача 5.

Под конец выполнения лабы в моем дереве уже есть 13 функций, чем я искренне горжусь.

Правда, большая часть откровенно плохи и на них смотреть больно, но…. Я только училась и это аргумент.

Забавно как большая часть своего же кода начинает ужасать через пару дней, когда уже становится более понятно что вообще происходит, но я, если честно, сейчас не очень в состоянии все это переделывать. Только хуже станет скорее всего…

В общем добавляем мы с помощью функции поиска. Ищем корень после которого нодик может встать и к нему и лепим влево или вправо. Сам же find работает рекурсивно, тоже весьма наивно и просто, если искомое меньше значения настоящего элемента, то идем влево, иначе вправо. Если же равно, то мы элемент нашли. А если элемента нет, то вернется последний посещенный корень, к которому мы как раз можем прикрепить новенький искомый.

Удаление у нас может быть трех видов: если у элемента нет детей, если один ребенок и если два ребенка. Если детей нет, то мы просто удаляем ссылку на этот элемент у родителя и больше ничего не делаем.

Если Ребенок один, то просто поднимаем этого ребенка на место удаляемого элемента, меняем ссылки и радуемся жизни.

Если же мы удаляем элемент с двумя детьми, то начинаются танцы с бубном под названием “найдите им приемного родителя пж”. Приемным родителем будет как раз таки наименьший

элемент (самый левый) из правого поддерева удаляемого элемента. Находим, удаляем из конца, вставляем на место удаляемого меняем ссылки. Все.

Следующие две функции — это сущий ад. Просто адское месиво состоящее из сотни условий которые появлялись на дебаге разных деревьев. Если смотреть сам файл, видно, сколько я страдала…

Если бы гарантировалось, что данный нам элемент есть в дереве, то все было бы куда проще. А проблемы возникают как раз с тем, что это число неизвестно где лежит. Я написала алгоритм для поиска элемента по типу find и дальше смотрела где он у меня падает и прописывала условия. Получилось вроде как три основных условия, равно искомому, больше или меньше, и на них уже все основано дальше. В зависимости от ситуации возвращаем или родителя или ребенка.

Оч сложно, сейчас понимаю, что в декартаче это было бы в 100 раз проще сделать.

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



class BinaryTree:
    def __init__(self):
        self.root = None
        self.num_nodes = 0


    def insert(self, new):
        if self.root == None:
            new_root = Node(new)
            self.root = new_root
            self.num_nodes += 1
            return

        new_node = Node(new)
        whether_found, parent = self.find(new)
        parent_value = parent.data
        new_node.parent = parent
        if new > parent_value:
            parent.right = new_node
            self.num_nodes += 1
            return
        parent.left = new_node
```

```
        self.num_nodes += 1
        return


    def find(self, query, rut='aa'):
        if rut == 'aa':
            rut = self.root
        if rut.data == query:
            return True, rut
        elif query < rut.data and rut.left != None:
            return self.find(query, rut.left)
        elif query > rut.data and rut.right != None:
            return self.find(query, rut.right)
        else:
            return(False, rut)


    def delete(self, query):
        whether_found, rut = self.find(query)

        # If there is no such node
        if whether_found == False:
            print('none')
            return

        # If the node to delete has no chindren
        if rut.left == None and rut.left == None:
            parent_value = rut.parent.data
            parent = rut.parent
            if query < parent_value:
                print(rut.parent.left)
                rut.parent.left = None
                self.num_nodes -= 1
                return rut
            rut.parent.right = None
            self.num_nodes -= 1
            return rut

        # If the node has only one child
        if rut.left != None and rut.right == None:
            parent_value = rut.parent.data
            if query < parent_value:
                rut.parent.left = rut.left
                self.num_nodes -= 1
                return
            rut.parent.right = rut.left
            self.num_nodes -= 1
            return
        elif rut.left == None and rut.right != None:
            parent_value = rut.parent.data
            if query < parent_value:
```

```
                rut.parent.left = rut.right
                self.num_nodes -= 1
                return
            rut.parent.right = rut.right
            self.num_nodes -= 1
            return

        # If the node has 2 children
        if rut.left and rut.right:
            minimum = self.find_min(rut.right)
            whether_found, node = self.find(minimum)
            before_node = node.parent
            if before_node.left.data == node.data:
                before_node.left = None
            else:
                before_node.right = None
            node.right = rut.right
            node.left = rut.left
            parent = rut.parent
            self.num_nodes -= 1

            if parent == None:
                self.root = node
                return
            node.parent = parent
            if rut.data == parent.data:
                parent.left = node
                return
            parent.right = node
            return


    def next_num(self, num, rut='aa'):
        if rut == 'aa':
            rut = self.root
        if rut == None:
            return 'none'
        if rut.left == None and rut.right == None and rut.data <= num:
            return 'none'
        elif rut.data == num:
            if rut.right != None:
                return self.next_num(num, rut.right)
            else:
                if rut.parent.data < num:
                    return rut.parent.data
                else:
                    return 'none'
        elif rut.data > num:
            if (rut.data - num) == 1:
                return rut.data
            if rut.left == None:
```

```
                return rut.data
            if rut.left.data == num:
                if rut.left.right == None:
                    return rut.data
                else:
                    return self.next_num(num, rut.left.right)
            if rut.left.data:
                return self.next_num(num, rut.left)
        elif rut.data < num:
            return self.next_num(num, rut.right)
        else:
            return self.next_num(num, rut.left)


    def prev_num(self, num, rut='aa'):
        if rut == 'aa':
            rut = self.root
        if rut == None:
            return 'none'
        if rut.left == None and rut.right == None and rut.data > num:
            return 'none'
        elif rut.data == num:
            if rut.left != None:
                return self.prev_num(num, rut.left)
            else:
                if rut.parent.data < num:
                    return rut.parent.data
                else:
                    return 'none'
        if rut.data < num:
            if (num - rut.data) == 1:
                return rut.data
            if rut.right == None:
                return rut.data
            if rut.right.data == num:
                if rut.right.left == None:
                    return rut.data
                else:
                    return self.prev_num(num, rut.right.left)
            if rut.right.data:
                return self.prev_num(num, rut.right)
        if rut.data > num:
            if rut.left.data == num:
                if rut.left.left:
                    return self.prev_num(num, rut.left)
                else:
                    if rut.parent.data < num:
                        return rut.parent.data
                    else:
                        return 'none'
            return self.prev_num(num, rut.left)
```

```
        else:
            return self.prev_num(num, rut.right)


    def find_min(self, rut):
        if rut.left:
            min_cur = rut.left.data
            next_rut = rut.left
            minimum = min(min_cur, self.find_min(next_rut))
            return minimum
        if not rut.right or rut.left:
            return(rut.data)
```

Задача 6.

Вроде все просто, просто берешь и проверяешь является ли данное дерево бинарным деревом поиска или нет.

Тут функция основана на том, что мы для каждой вершинки проверяем, меньше ли наибольшее число в левом поддереве чем она сама и больше ли наименьшее в правом. Поиск максимума и минимума какой-то кривой, но вроде работает и ладно. Было бы проще, если бы мы 100% знали, что это бинарник, но смысл тогда в задаче?)))

Самым, однако, сложным, для меня оказалось построить дерево. Я вроде как и понимаю что к чему зачем и как, ко как это компуктеру-то моему несчастному объяснить… Не придумала ничего лучше чем читать файл два раза, первый раз создавая вершинки с порядком добавления, а второй раз уже для считывания. Можно и один раз, но я пока не догадалась как.

```
# I honestly hate the way the way adding nodes is described, I have no idea
# how it could be done easier,
# but this is painful save me pls I wanna cryyyyyyyyy
import math

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


class Tree():
    def __init__(self) -> None:
        self.root = None


    def insert(self, nodes, rut, line):
```

```
        if line[0] == rut.data:
            if line[1] != -1:
                rut.left = nodes[line[1]]
            if line[2] != -1:
                rut.right = nodes[line[2]]
            return True
        if_inserted = False
        if rut.left != None:
            if_inserted = self.insert(nodes, rut.left, line)
        if if_inserted == True:
            return True
        if rut.right != None:
            return self.insert(nodes, rut.right, line)


    def check_bst(self, rut):
        if rut.left != None:
            if rut.left.data < rut.data:
                if self.max_subtree(rut.left, -math.inf) > rut.data:
                    return 'INCORRECT'
                self.check_bst(rut.left)
            else:
                return 'INCORRECT'
        if rut.right != None:
            if rut.right.data > rut.data:
                if self.min_subtree(rut.right, math.inf) < rut.data:
                    return 'INCORRECT'
                self.check_bst(rut.right)
            else:
                return 'INCORRECT'
        return 'CORRECT'


    def max_subtree(self, rut):
        biggest = -math.inf
        tree = rut
        while tree.left != None:
            if tree.left.data > biggest:
                biggest = tree.left.data
            tree = tree.left
            tree_ = tree
            while tree.right is not None:
                if tree.right.data > biggest:
                    biggest = tree.right.data
                tree = tree.right
            tree = tree_
        tree = rut
        while tree.right != None:
            if tree.right.data > biggest:
                biggest = tree.right.data
            tree = tree.right
```

```
            tree_ = tree
            while tree.left is not None:
                if tree.left.data > biggest:
                    biggest = tree.left.data
                tree = tree.left
            tree = tree_
        return biggest


    def min_subtree(self, rut):
        smallest = math.inf
        tree = rut
        while tree.left != None:
            if tree.left.data < smallest:
                smallest = tree.left.data
            tree = tree.left
            tree_ = tree
            while tree.right is not None:
                if tree.right.data < smallest:
                    smallest = tree.right.data
                tree = tree.right
            tree = tree_
        tree = rut
        while tree.right != None:
            if tree.right.data < smallest:
                smallest = tree.right.data
            tree = tree.right
            tree_ = tree
            while tree.left is not None:
                if tree.left.data < smallest:
                    smallest = tree.left.data
                tree = tree.left
            tree = tree_
        return smallest


    def printtree(self, rut='aa'):
        if rut == 'aa':
            rut = self.root

        to_print = ''
        if rut.left:
            to_print += str(rut.left.data)
        else:
            to_print += 'None'

        to_print += '|' + str(rut.data) + '|'

        if rut.right:
            to_print += str(rut.right.data)
        else:
```

```
            to_print += 'None'
        print(to_print)

        if rut.left:
            self.printtree(rut.left)
        if rut.right:
            self.printtree(rut.right)
        return




my_tree = Tree()

with open('input_6.txt') as f:
    num_nodes = int(f.readline())
    nodes = {}
    for i in range(num_nodes):
        data, id_left_ch, id_right_ch = list(map(int, f.readline().split()))
        if i == 0:
            new_root = Node(data=data)
            my_tree.root = new_root
        else:
            new_node = Node(data=data)
            nodes[i] = new_node

with open('input_6.txt') as f:
    num_nodes = int(f.readline())
    for i in range(num_nodes):
        line = list(map(int, f.readline().split()))
        my_tree.insert(nodes, my_tree.root, line)

with open('output_6.txt', 'w') as d:
    if num_nodes == 0:
        d.write('CORRECT')
    else:
        d.write(my_tree.check_bst(my_tree.root))
```

Задача 7.

Наверное, задача сложная, если 6 была бы реализована как-то иначе, но Тут мне было нужно буквально добавить два знака равно на весь код. Я даже не буду копировать всю эту скатерть кода, только измененную часть)))

В общем задача бЕЕЕЕЕЕЕЕЕЕсплатная, приятненько)))))))))

```
    def check_bst(self, rut):
        if rut.left != None:
            if rut.left.data < rut.data:
                if self.max_subtree(rut.left) >= rut.data:
```

```
                    return 'INCORRECT'
                self.check_bst(rut.left)
            else:
                return 'INCORRECT'
        if rut.right != None:
            if rut.right.data >= rut.data:
                if self.min_subtree(rut.right) < rut.data:
                    return 'INCORRECT'
                self.check_bst(rut.right)
            else:
                return 'INCORRECT'
        return 'CORRECT'


    def max_subtree(self, rut):
        biggest = -math.inf
        tree = rut
        while tree.left != None:
            if tree.left.data >= biggest:
                biggest = tree.left.data
            tree = tree.left
            tree_ = tree
            while tree.right is not None:
                if tree.right.data > biggest:
                    biggest = tree.right.data
                tree = tree.right
            tree = tree_
        tree = rut
        while tree.right != None:
            if tree.right.data >= biggest:
                biggest = tree.right.data
            tree = tree.right
            tree_ = tree
            while tree.left is not None:
                if tree.left.data > biggest:
                    biggest = tree.left.data
                tree = tree.left
            tree = tree_
        return biggest
```

Задача 8.

Зачем писать новый класс если есть уже полный файл с реализованными основными функциями? Тем более ведь классно иметь многофункциональную штуку в одном файле, и чтобы оно работало, правда?) Ведь я теперь даже пользоваться этим могу, так приятно.

В общем я просто дописала еще две функции, одна начинает обход, другая уходит вглубь. Весь код не вижу смысла вставлять, просто вот новенькие функции:

```
    def depth(self, rut='aa'):
        count = 0
        left = 0
        right = 0
        if rut == 'aa':
            rut = self.root
        if rut == None:
            return 0
        if rut != None and rut.left == None and rut.right == None:
            return 1
        if rut.left != None:
            left = 1 + self.go_inside(rut.left)
        if rut.right != None:
            right = 1 + self.go_inside(rut.right)
        count += max(left, right)
        if count == 0:
            return 0
        else:
            return count + 1


    def go_inside(self, rut):
        left = 0
        right = 0
        if rut.left == None and rut.right == None:
            return 0
        if rut.left != None:
            left = 1 + self.go_inside(rut.left)
        if rut.right != None:
            right = 1 + self.go_inside(rut.right)
        return max(left, right)
```

Задача 9.

Удаление поддеревьев??? После того как я написала полный класс с удалением нодов у которых два ребенка???? Серьезно??????

Да это ж буквально найти ячейку и удалить связь у ее родителя с ней…. И все!

Искренне рада еще одной бесплатной задаче :D

Опять-таки добавляю просто одну микрофункцию в мой макрокласс и все))

```
    def delete_subtree(self, node_id):
        whether_found, node = self.find(node_id)
        if whether_found == False:
            return self.num_nodes
        parent = node.parent
        if parent.data > node.data:
```

```
            diff = self.count_nodes(node)
            self.num_nodes -= diff
            parent.left = None
            return self.num_nodes
        elif parent.data < node.data:
            diff = self.count_nodes(node)
            self.num_nodes -= diff
            parent.right = None
            return self.num_nodes
    def count_nodes(self, node='aa'):
        if node == 'aa':
            node = self.root
        if node is None:
            return 0
        return 1 + self.count_nodes(node.left) + self.count_nodes(node.right)
```

Задача 16.

Вот мне повезло описывать эту функцию самой последней, потому что теперь-то я знаю, что гораздо проще было бы ее написать не на основе моего бинарного дерева, а на основе декартова дерева.

Но это сейчас оно у меня уже написано так, что это можно сделать 8 строчек. А так как за декартач я села только после 16, как раз этой, задачи, я не знала что можно по-другому)))

В кратце, берем дерево, обходим его в обратном порядке, поднимаемся наверх, если нет левых детей, обходим у родителя всех левых детей и тд. И тп.

Достаточно красиво, неплохо, робит, цветет и пахнет)))

```
    def find_kth_max(self, k):
        rut = self.root
        while rut.right != None:
            rut = rut.right
        max_decreased = []
        max_decreased.append(rut.data)
        current_max = 1
        returned = self.kth_max_assistant(rut, k, current_max)
        current_max = returned[1]
        max_decreased += returned[0]
        while current_max < k:
            if rut.parent:
                rut = rut.parent
                max_decreased.append(rut.data)
                returned = self.kth_max_assistant(rut, k, current_max)
                current_max = returned[1]
                max_decreased += returned[0]
            elif rut.left:
```

```
                max_decreased.append(rut.left.data)
                returned = self.kth_max_assistant(rut, k, current_max)
                current_max = returned[1]
                max_decreased += returned[0]
        return max_decreased[k - 1]


    def kth_max_assistant(self, rut, k, current_max, kth=1):
        to_add = []
        if rut.right == None and rut.left == None:
            return [to_add, current_max]
        if rut.right != None and kth != 1:
            current_max += 1
            returned = self.kth_max_assistant(rut.right, k, current_max, kth + 1)
            to_add += returned[0]
            to_add.append(rut.right.data)
            current_max = returned[1]
        if rut.left != None:
            current_max += 1
            returned = self.kth_max_assistant(rut.left, k, current_max, kth + 1)
            to_add += returned[0]
            to_add.append(rut.left.data)
            current_max = returned[1]
        return [to_add, current_max]
```

Описание проведенных тестов.

3

Input	Output
+ 1 + 3 + 3 > 1 > 2 > 3 + 2 > 1	3 3 0 2

4

Input	Output
+ 1 + 4 + 3 + 3 ? 1 ? 2 ? 3 + 2 ? 3	1 3 4 3

5

Input	Output
insert 2 insert 5 insert 3 exists 2 exists 4 next 4 prev 4 delete 5 next 4 prev 4	True False 5 3 none 3

6

Input	Output
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT
0	CORRECT
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT
7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT
4 4 1 -1 2 2 3 1 -1 -1 5 -1 -1	INCORRECT

7

Input	Output
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT
3 2 1 2 1 -1 -1 2 -1 -1	CORRECT
3 2 1 2 2 -1 -1 3 -1 -1	INCORRECT
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT
7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT
1 2147483647 -1 -1	CORRECT

8

Ваш код завершился с ошибкой, тест 24
Подсказка: это дерево размера 3156, в котором бывают только левые потомки.

Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		0.171	10604544	38820	3
1	OK	0.031	9314304	46	1
2	OK	0.015	9285632	3	1
3	OK	0.031	9289728	11	1
4	OK	0.031	9256960	18	1
5	OK	0.031	9281536	103	1
6	OK	0.031	9318400	76	2
7	OK	0.031	9318400	155	2
8	OK	0.031	9256960	163	2
9	OK	0.031	9310208	57	1
10	OK	0.015	9326592	161	1
11	OK	0.031	9277440	2099	1
12	OK	0.015	9302016	1197	3
13	OK	0.031	9322496	2073	3
14	OK	0.031	9342976	2139	3
15	OK	0.015	9273344	686	1
16	OK	0.031	9277440	2128	2
17	OK	0.031	9314304	8777	1
18	OK	0.156	10448896	10426	3
19	OK	0.171	10457088	16336	3
20	OK	0.171	10432512	16835	3
21	OK	0.031	9261056	3520	1
22	OK	0.031	9355264	16969	2
23	OK	0.031	9584640	36534	2
24	RE	0.171	10604544	38820	0

9

Превышение ограничения на время работы, тест 19
Подсказка: это случайное сбалансированное дерево с 198184 вершинами, из которого в случайном порядке удаляются все элементы, кроме к орня, включая уже несуществующие на момент удаления элементы.

Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		Превышение	80306176	6007604	1077960
1	OK	0.015	9392128	58	12
2	OK	0.031	9396224	27	12
3	OK	0.015	9396224	34	15
4	OK	0.031	9392128	211	30
5	OK	0.031	9383936	246	30
6	OK	0.031	9441280	3437	457
7	OK	0.031	9400320	3363	483
8	OK	0.031	9445376	18842	4247
9	OK	0.031	9465856	25683	3739
10	OK	0.046	10153984	69351	14791
11	OK	0.046	10326016	88936	11629
12	OK	0.093	12087296	244892	40297
13	OK	0.140	12201984	255614	37596
14	OK	0.296	21213184	978616	141281
15	OK	0.312	21106688	992647	137802
16	OK	0.937	39649280	2488583	634135
17	OK	1.093	50327552	3489729	483105
18	OK	1.765	64368640	4639039	1077960
19	TL	Превышение	80306176	6007604	931260

16

Input	Output
11 +1 5 +1 3 +1 7 0 1 0 2 0 3 -1 5 +1 10 0 1 0 2 0 3	7 5 3 10 7 3

Выводы по проделанной работе.

Я наконец-то доделала эту лабу, она заняла у меня 4,5 дня и сколько-то там ночей что были между этими днями.

Что я могу сказать, деревья — очень красивая вещь. Особенно декартово. Я еще не так много его прописала, но точно займусь этим после сессии. Также хочется написать хоть разочек splay, потому что сейчас я понимала, что банально не хватит времени и все.

Обычное дерево у меня вышло в 13 функций и я в шоке, потому что в прошлом семестре я пыталась написать обычный стек, и у меня это занимало 3 дня, а сейчас я пишу вот это??? Сама???

В общем я дописала и счастлива, можно теперь и к графам переходить))))