САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ

ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №2

по курсу «Алгоритмы и структуры данных»

Тема: Двоичные деревья поиска.

Вариант 29

Выполнил:

Логачев Д.С.

К3139/К32402

Проверила:

Артамонова В.Е.

Санкт-Петербург

2022 г.

# Содержание отчета

[Содержание отчета 2](#_Toc115807389)

[Задачи по варианту 3](#_Toc115807390)

[4 задача. Простейший неявный ключ (1 балл) 3](#_Toc115807391)

[12 задача. Проверка сбалансированности (2 балла) 5](#_Toc115807392)

[15 задача. Удаление из АВЛ-дерева (3 балл) 7](#_Toc115807393)

[Вывод 14](#_Toc115807394)

# Задачи по варианту

## 4 задача. Простейший неявный ключ (1 балл)

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

«+ x» – добавить в дерево x (если x уже есть, ничего не делать).

«? k» – вернуть k-й по возрастанию элемент.

class node:

def \_\_init\_\_(self):

self.key = None

self.left = None

self.right = None

self.parent = None

self.size = None

class binTree:

def \_\_init\_\_(self):

self.root = None

def search(self, data, root):

if root is None or data == root.key:

return root

if data < root.key:

if root.left is not None:

return self.search(data, root.left)

return root

if root.right is not None:

return self.search(data, root.right)

return root

def insert(self, data):

if self.root is None:

self.root = node()

self.root.key = data

self.root.size = 1

else:

t = self.search(data, self.root)

if t.key == data:

return

elif data < t.key:

t.left = node()

t.left.key = data

t.left.parent = t

t.left.size = 1

else:

t.right = node()

t.right.key = data

t.right.parent = t

t.right.size = 1

while t is not None:

t.size += 1

t = t.parent

def kMax(self, root, k):

if root.left is None:

s = 0

else:

s = root.left.size

if k == s + 1:

return root.key

if k < s + 1:

return self.kMax(root.left, k)

return self.kMax(root.right, k-s-1)

f = open('input.txt', 'r')

o = open("output.txt", "w")

L = f.readlines()

T = binTree()

for i in L:

com, x = i.split()

if com == '+':

T.insert(int(x))

else:

o.write(str(T.kMax(T.root, int(x))) + "\n")

В данном решении используем функции поиска, вставки и нахождения узла с максимальным значениям ключа, основываясь на псевдокодах из лекции. В основной части кода применяем данные функции для вставки элементов и возвращения элементов двоичного дерева. 

## 12 задача. Проверка сбалансированности (2 балла)

АВЛ-дерево является сбалансированным в следующем смысле: для любой вершины высота ее левого поддерева отличается от высоты ее правого поддерева не больше, чем на единицу. Введем понятие баланса вершины: для вершины дерева V ее баланс B(V) равен разности высоты правого поддерева и высоты левого поддерева. Таким образом, свойство АВЛ-дерева, приведенное выше, можно сформулировать следующим образом: для любой ее вершины V выполняется следующее неравенство:

−1 ≤ B(V ) ≤ 1

Обратите внимание, что, по историческим причинам, определение баланса в этой и последующих задачах этой недели «зеркально отражено» по сравнению с определением баланса в лекциях! Надеемся, что этот факт не доставит Вам неудобств. В литературе по алгоритмам – как российской, так и мировой – ситуация, как правило, примерно та же.

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

def build\_tree\_from\_input(array):

for i in range(len(array)):

if array[i][1]:

array[array[i][1] - 1] += [i, None]

if array[i][2]:

array[array[i][2] - 1] += [i, None]

array[0] += [None, None]

return array

def count\_height(array, i):

height = 1

while i:

if (not array[i][4]) or (array[i][4] and array[i][4] < height):

array[i][4] = height

height += 1

i = array[i][3]

else:

break

return array

def balance\_tree(array):

for i in range(len(array)):

if not array[i][1] and not array[i][2]:

array = count\_height(array, i)

for i in array:

left = 0

right = 0

if i[1]:

left = array[i[1] - 1][4]

if i[2]:

right = array[i[2] - 1][4]

output.write(str(right - left) + "\n")

file = open("input.txt", "r")

output = open("output.txt", "w")

data = file.readlines()

data = data[1:]

input\_data = []

for i in data:

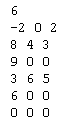
input\_data.append(list(map(int, i.split())))

balance\_tree(build\_tree\_from\_input(input\_data))

file.close()

output.close()

Создадим список, который будет описывать поведение нашего дерева (создадим дерево в виде списка): у нас есть базовый узел типа list в котором есть 2 подлиста, так как нам известны индексы, мы спокойно можем построить дерево таким образом. Остается лишь посчитать высоту левого и правого поддерева и найти их разницу что и будет балансом дерева.



## 15 задача. Удаление из АВЛ-дерева (3 балл)

Удаление из АВЛ-дерева вершины с ключом X, при условии ее наличия, осуществляется следующим образом:

• путем спуска от корня и проверки ключей находится V – удаляемая вершина;

• если вершина V – лист (то есть, у нее нет детей):

– удаляем вершину;

– поднимаемся к корню, начиная с бывшего родителя вершины V , при этом если встречается несбалансиро-

ванная вершина, то производим поворот.

• если у вершины V не существует левого ребенка:

– следовательно, баланс вершины равен единице и ее правый ребенок – лист;

– заменяем вершину V ее правым ребенком;

– поднимаемся к корню, производя, где необходимо, балансировку.

• иначе:

– находим R – самую правую вершину в левом поддереве;

– переносим ключ вершины R в вершину V;

– удаляем вершину R (у нее нет правого ребенка, поэтому она либо лист, либо имеет левого ребенка, являю-

щегося листом);

– поднимаемся к корню, начиная с бывшего родителя вершины R, производя балансировку.

Исключением является случай, когда производится удаление из дерева, состоящего из одной вершины - корня. Результатом удаления в этом случае будет пустое дерево. Указанный алгоритм не является единственно возможным, но мы просим Вас реализовать именно его, так как тестирующая система проверяет точное равенство получающихся деревьев.

import sys

from collections import deque

res = []

class Node:

def \_\_init\_\_(self, data):

self.data = data

self.par = None

self.left = None

self.right = None

self.height = -1

self.id = 0

self.next = None

def build\_tree(root):

if (root.left != None):

build\_tree(root.left)

if (root.right != None):

build\_tree(root.right)

fix\_height(root)

def height\_right(root):

if (root.right == None):

return 0

return root.right.height

def height\_left(root):

if (root.left == None):

return 0

return root.left.height

def fix\_height(root):

root.height = max(height\_left(root), height\_right(root)) + 1

def get\_balance(root):

r = 0

l = 0

if (root.right != None):

r = root.right.height

if (root.left != None):

l = root.left.height

return r - l

def rotate\_tree(node, side):

u = None

if (side == 'left'):

if node is None or node.right is None:

return node

parent = node.par

right = node.right

right\_left = right.left

if parent:

if parent.right == node:

parent.right = right

else:

parent.left = right

right.par = parent

right.left = node

node.par = right

node.right = right\_left

if right\_left:

right\_left.par = node

fix\_height(node)

fix\_height(right)

return right

else:

if node is None or node.left is None:

return node

parent = node.par

left = node.left

left\_right = left.right

if parent:

if parent.left == node:

parent.left = left

else:

parent.right = left

left.par = parent

left.right = node

node.par = left

node.left = left\_right

if left\_right:

left\_right.par = node

fix\_height(node)

fix\_height(left)

return left

def getMax(root):

if (root == None):

return root

while root.right != None:

root = root.right

return root

def blnc(root):

fix\_height(root)

balance = get\_balance(root)

if balance > 1:

if get\_balance(root.right) < 0:

root.right = rotate\_tree(root.right, 'right')

return rotate\_tree(root, 'left')

elif balance < -1:

if get\_balance(root.left) > 0:

root.left = rotate\_tree(root.left, 'left')

return rotate\_tree(root, 'right')

return root

def delete(root, key):

if root == None:

return root

elif key < root.data:

root.left = delete(root.left, key)

elif key > root.data:

root.right = delete(root.right, key)

else:

if root.left is None and root.right is None:

return None

if root.left == None:

root = root.right

return blnc(root)

temp = getMax(root.left)

root.data = temp.data

root.left = delete(root.left, temp.data)

return blnc(root)

def printBST(root, n):

global res

queue = deque()

queue.append((root, (-1, -1)))

while queue:

u, v = queue.popleft()

if (v[0] >= 0 and v[1] >= 0):

res[v[0]][v[1]] = len(res) + 1

if (u == None):

continue

tmp = [0, 0, 0]

tmp[0] = u.data

res.append(tmp)

cur = len(res)

if (u.left != None):

queue.append((u.left, (cur - 1, 1)))

if (u.right != None):

queue.append((u.right, (cur - 1, 2)))

sys.stdin = open("input.txt", "r")

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

input = sys.stdin.readline

n = int(input())

root = []

for i in range(n + 10):

root.append(Node(0))

for i in range(n):

k, l, r = map(int, input().split())

root[i + 1].data = k

if (l):

root[i + 1].left = root[l]

root[l].par = root[i + 1]

if (r):

root[i + 1].right = root[r]

root[r].par = root[i + 1]

val = int(input())

build\_tree(root[1])

root[1] = delete(root[1], val)

printBST(root[1], n)

sys.stdout.write(str(len(res)) + "\n")

n = len(res)

for i, j, k in res:

sys.stdout.write(str(i) + ' ' + str(j) + ' ' + str(k) + '\n')

Принцип построения очень похож с BST , за исключением того, что теперь надо реализовать функцию проверки баланса(высоты левого и правого поддеревьев не должны отличаться на >1 ) и функции поворотов, существует 4 варианта поворота, маленький правый, маленький левый, большой правый, большой левый. Входные данные содержат индекс у узлов, а не сами узлы. Считывая данные, мы переходим к строительству нашего дерева методом build. При удалении мы смотрим на наш баланс и в случае расхождений смотрим на все те же 4 случая, затем выбираем вращение и балансируем дерево.

# Вывод

В данной лабораторной работе были реализованы алгоритмы,

представляющие двоичные деревья поиска, а также их разновидности, различные обходы деревьев и алгоритмы, позволяющие модифицировать деревья в зависимости от поставленной задачи.

Итого, для определенного класса задач двоичные деревья поиска ‒ очень

полезная структура данных, обеспечивающая простоту вычислений и

позволяющая уделять меньше времени на выполнение ряда операций.