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 17

Выполнил:

Прокопец С.Р.

К3139

Проверила:

Артамонова В.Е.

Санкт-Петербург

2022 г.

# Содержание отчета

[Содержание отчета 2](#_Toc128645006)

[Задачи по варианту 2](#_Toc128645007)

[Задача №2. Гирлянда [2 s, 256 Mb, 1 балл] 2](#_Toc128645008)

[Задача №8. Высота дерева возвращается [2 s, 256 Mb, 2 балла] 4](#_Toc128645009)

[Задача №17. Множество с суммой [120 s, 512 Mb, 3 балла] 4](#_Toc128645010)

[Дополнительные задачи 9](#_Toc128645011)

[Задача №1. Обход двоичного дерева [5 s, 512 Mb, 1 балл] 9](#_Toc128645012)

[Задача №3 Простейшее BST [2 s, 256 Mb, 1 балл] 10](#_Toc128645013)

[Задача №4 Простейший неявный ключ [2 s, 256 Mb, 1 балл] 11](#_Toc128645014)

[Задача №5 Простое двоичное дерево поиска [2 s, 512 Mb, 1 балл] 12](#_Toc128645015)

[Задача №6 Опознание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла] 15](#_Toc128645016)

[Задача №7 Опознание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла] 17](#_Toc128645017)

[Задача №10 Проверка корректности [2 s, 256 Mb, 2 балла] 18](#_Toc128645018)

[Задача №11 Сбалансированное двоичное дерево поиска [2 s, 512 Mb, 2 балла] 19](#_Toc128645019)

[Задача №12 Вставка в АВЛ-дерево [2 s, 256 Mb, 3 балла] 23](#_Toc128645020)

[Задача №15 Удаление из АВЛ-дерева [2 s, 256 Mb, 3 балла] 26](#_Toc128645021)

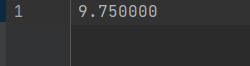
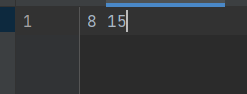
# Задачи по варианту

## Задача №2. Гирлянда [2 s, 256 Mb, 1 балл]

Требуется найти минимальное значение высоты второго конца B гирлянды

filein = open("input.txt")  
n, height = filein.readline().split()  
n = int(n)  
height = float(height)  
filein.close()  
  
error = 0.1 \*\* 10  
  
  
def equal(a, b):  
 return abs(a - b) <= error  
  
  
def less(a, b):  
 return (a < b) and (not equal(a, b))  
  
  
def more(a, b):  
 return (a > b) and (not equal(a, b))  
  
  
heights = [0] \* n  
heights[0] = height  
res = 1000000000  
left = 0  
right = heights[0]  
  
while less(left, right):  
 heights[1] = (left + right) / 2  
 heights[-1] = 0  
 isUp = False  
 for i in range(2, n):  
 heights[i] = 2 \* heights[i-1] - heights[i-2] + 2  
 if not more(heights[i], 0):  
 isUp = True  
 break  
 if more(heights[-1], 0):  
 res = min(res, heights[-1])  
 if isUp:  
 left = heights[1]  
 else:  
 right = heights[1]  
  
fileout = open("output.txt", "w")  
fileout.write("%.6f" % res)  
fileout.close()

В решении используется двоичный поиск для установки значения лампочки, идущей за первой, из уравнения высоты каждой лампочки выражено значение высоты правой лампочки и вычисляется в цикле на основании значений двух левых лампочек.

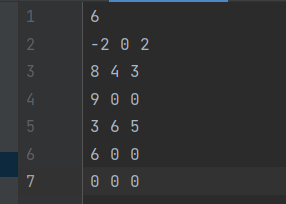
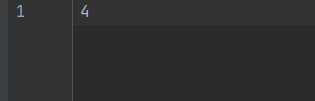


## Задача №8. Высота дерева возвращается [2 s, 256 Mb, 2 балла]

Нужно найти высоту двоичного дерева

inp = open("input.txt")  
  
n = int(inp.readline())  
woods = []  
deeps = [0] \* (n + 1)  
  
for \_ in range(n):  
 value, left, right = map(int, inp.readline().split())  
 woods.append((left, right))  
  
inp.close()  
  
# starts from leaves  
for i in range(n - 1, -1, -1):  
 if (woods[i][0] == 0) and (woods[i][1] == 0):  
 deeps[i + 1] = 1  
 else:  
 deeps[i + 1] = max(deeps[woods[i][0]], deeps[woods[i][1]]) + 1  
  
out = open("output.txt", "w")  
out.write(str(deeps[1]) if n > 0 else "0")  
out.close()

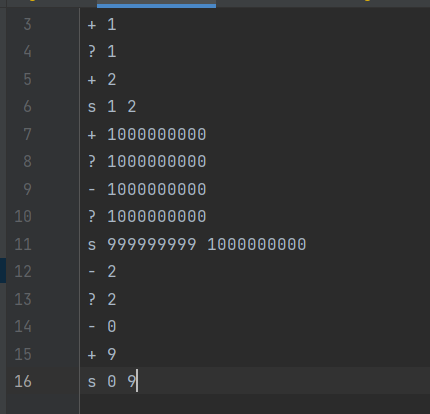
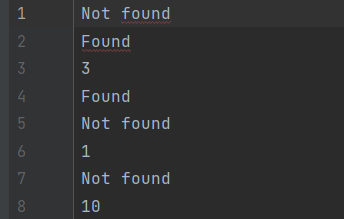
Простым перебором находим высоту двоичного дерева.

## Задача №17. Множество с суммой [120 s, 512 Mb, 3 балла]

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

from collections import deque  
  
  
class Node:  
 def \_\_init\_\_(self, key, left=None, right=None, parent=None):  
 self.left = left  
 self.right = right  
 self.parent = parent  
 self.key = key  
  
  
class SplayTree:  
 def set\_parent(self, child, parent):  
 if child is not None:  
 child.parent = parent  
  
 def keep\_parent(self, node):  
 self.set\_parent(node.left, node)  
 self.set\_parent(node.right, node)  
  
 def rotate(self, parent, child):  
 grandparent = parent.parent  
 if grandparent is not None:  
 if grandparent.left == parent:  
 grandparent.left = child  
 else:  
 grandparent.right = child  
  
 if parent.left == child:  
 parent.left, child.right = child.right, parent  
 else:  
 parent.right, child.left = child.left, parent  
  
 self.keep\_parent(child)  
 self.keep\_parent(parent)  
 child.parent = grandparent  
  
 def splay(self, node):  
 if node.parent is None:  
 return node  
 parent = node.parent  
 grandparent = parent.parent  
 if grandparent is None:  
 self.rotate(parent, node)  
 return node  
 else:  
 zigzig = (grandparent.left == parent) == (parent.left == node)  
 if zigzig:  
 self.rotate(grandparent, parent)  
 self.rotate(parent, node)  
 else:  
 self.rotate(parent, node)  
 self.rotate(grandparent, node)  
 return self.splay(node)  
  
 def find(self, node, key):  
 if node is None:  
 return None  
 if key == node.key:  
 return self.splay(node)  
 if key < node.key and node.left is not None:  
 return self.find(node.left, key)  
 if key > node.key and node.right is not None:  
 return self.find(node.right, key)  
 return self.splay(node)  
  
 def split(self, root, key):  
 if root is None:  
 return None, None  
 root = self.find(root, key)  
 if root.key == key:  
 self.set\_parent(root.left, None)  
 self.set\_parent(root.right, None)  
 return root.left, root.right  
 if root.key < key:  
 right, root.right = root.right, None  
 self.set\_parent(right, None)  
 return root, right  
 else:  
 left, root.left = root.left, None  
 self.set\_parent(left, None)  
 return left, root  
  
 def insert(self, root, key):  
 left, right = self.split(root, key)  
 root = Node(key, left, right)  
 self.keep\_parent(root)  
 return root  
  
 def merge(self, left, right):  
 if right is None:  
 return left  
 if left is None:  
 return right  
 right = self.find(right, left.key)  
 right.left, left.parent = left, right  
 return right  
  
 def remove(self, root, key):  
 root = self.find(root, key)  
 self.set\_parent(root.left, None)  
 self.set\_parent(root.right, None)  
 return self.merge(root.left, root.right)  
  
 def sum(self, root, l, r):  
 stack = deque()  
 nums = []  
 if root is None:  
 return 0  
 while True:  
 stack.append(root)  
 if (root.left is None) or (root.key <= l):  
 break  
 root = root.left  
 while True:  
 if len(stack) != 0:  
 nums.append(stack[-1].key)  
 if (stack[-1].right is not None) and (stack[-1].key <= r):  
 root = stack.pop().right  
 else:  
 stack.pop()  
 continue  
 while True:  
 stack.append(root)  
 if (root.left is None) or (root.key <= l):  
 break  
 root = root.left  
 if len(stack) == 0:  
 summ = 0  
 for num in nums:  
 if (num >= l) and (num <= r):  
 summ += num  
 return summ  
  
  
M = 1000000001  
last\_op = 0  
  
inp = open("input.txt")  
out = open("output.txt", "w")  
n = int(inp.readline())  
  
root = None  
tree = SplayTree()  
  
for i in range(n):  
 line = inp.readline().split()  
 num = (int(line[1]) + last\_op) % M  
 if line[0] == "?":  
 root = tree.find(root, num)  
 if (root is not None) and (root.key == num):  
 out.write("Found\n")  
 else:  
 out.write("Not found\n")  
 elif line[0] == "+":  
 if root is None:  
 root = Node(num)  
 else:  
 root = tree.insert(root, num)  
 elif line[0] == "-":  
 if root is not None:  
 root = tree.remove(root, num)  
 elif line[0] == "s":  
 num\_s = (int(line[2]) + last\_op) % M  
 last\_op = tree.sum(root, num, num\_s)  
 out.write(f"{last\_op}\n")  
  
inp.close()  
out.close()

# Дополнительные задачи

## Задача №1. Обход двоичного дерева [5 s, 512 Mb, 1 балл]

import time  
  
t\_start = time.perf\_counter()  
import tracemalloc  
  
tracemalloc.start()  
  
from common.Node import Node  
from common.TreeBuilder import TreeBuilder  
  
  
def in\_order(tree: Node):  
 if tree is not None:  
 in\_order(tree.leftChild)  
 print(tree.node, end=" ")  
 in\_order(tree.rightChild)  
  
  
def pre\_order(tree: Node):  
 if tree is not None:  
 print(tree.node, end=" ")  
 pre\_order(tree.leftChild)  
 pre\_order(tree.rightChild)  
  
  
def post\_order(tree: Node):  
 if tree is not None:  
 post\_order(tree.leftChild)  
 post\_order(tree.rightChild)  
 print(tree.node, end=" ")  
  
f1 = open('input.txt')  
n = int(f1.readline())  
input\_data = []  
for line in f1:  
 k, l, r = map(int, line.split(" "))  
 input\_data.append([k, l, r])  
  
tree = TreeBuilder.buildBST(input\_data)  
in\_order(tree)  
print()  
pre\_order(tree)  
print()  
post\_order(tree)  
  
  
print("Время работы: %s секунд " % (time.perf\_counter() - t\_start))  
mc, mp = tracemalloc.get\_traced\_memory()  
print("Memory: current %f KiB, peak %f KiB" % (mc / 1024, mp / 1024))

## Задача №3 Простейшее BST [2 s, 256 Mb, 1 балл]

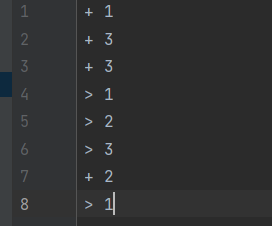
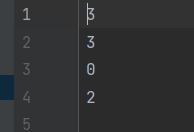
В этой задаче вам нужно написать простейшее BST по явному ключу и отвечать им на запросы:

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

«> x» – вернуть минимальный элемент больше x или 0, если таких неь

class Node:  
 def \_\_init\_\_(self, key):  
 self.left = None  
 self.right = None  
 self.key = key  
  
  
class Tree:  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, key):  
 parent = None  
 node = self.root  
 while node is not None:  
 parent = node  
 if key < node.key:  
 node = node.left  
 elif key > node.key:  
 node = node.right  
 else:  
 return  
 new = Node(key)  
 if parent is None:  
 self.root = new  
 elif key < parent.key:  
 parent.left = new  
 elif key > parent.key:  
 parent.right = new  
   
 def min(self, x):  
 if self.root is None:  
 return 0  
 way = []  
 node = self.root  
 while True:  
 way.append(node)  
 if x > node.key:  
 if node.right is None:  
 break  
 node = node.right  
 elif x < node.key:  
 if node.left is None:  
 return node.key  
 node = node.left  
 else:  
 if node.right is None:  
 break  
 node = node.right  
 while node.left is not None:  
 node = node.left  
 return node.key  
 for i in range(len(way) - 1, -1, -1):  
 if way[i].key > x:  
 return way[i].key  
 return 0  
  
  
filein = open("input.txt")  
fileout = open("output.txt", "w")  
  
tree = Tree()  
  
  
line = filein.readline()  
while line != "":  
 items = line.split()  
 if items[0] == "+":  
 tree.insert(int(items[1]))  
 else:  
 fileout.write(f"{tree.min(int(items[1]))}\n")  
 line = filein.readline()  
  
  
filein.close()  
fileout.close()

В поддереве производим поиск элемента большего чем данный

## Задача №4 Простейший неявный ключ [2 s, 256 Mb, 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.

f1 = open("input.txt", "r")  
f2 = open("output.txt", "w")  
import time  
start\_time = time.time()  
class BST:  
 def \_\_init\_\_(self):  
 self.mas = []  
  
 def binarySearch(self, arr, low, high, x):  
 while low <= high:  
 mid = low + (high - low) // 2  
 if arr[mid] == x:  
 return None  
 if arr[mid] == x:  
 return mid  
 elif arr[mid] < x:  
 high = mid - 1  
 else:  
 low = mid + 1  
 return high  
  
 def insert(self, val):  
 indexpos = self.binarySearch(self.mas, 0, len(self.mas) - 1, val)  
 if indexpos != None:  
 self.mas = self.mas[: indexpos + 1] + [val] + self.mas[indexpos + 1 :]  
  
bst = BST()  
answer = ""  
iter = 0  
for i in f1.readlines():  
 iter += 1  
 if iter % 1000 == 0:  
 print(str(iter) + "/300000")  
 cmd = i.split(" ")  
 v, cmd = cmd[1], cmd[0]  
  
 if cmd == "+" and int(v):  
 bst.insert(int(v))  
 if cmd == "?":  
 v = int(v)  
 answer += str(bst.mas[-v]) + "\n"  
f2.write(answer)  
  
print("--- %s seconds ---" % (time.time() - start\_time))

Двоичное дерево поиска позволяет эффективно решить поставленную задачу, если использовать бинарный поиск для созданий данного дерева

## Задача №5 Простое двоичное дерево поиска [2 s, 512 Mb, 1 балл]

Нудно реализовать простое двоичное дерево поиска.

class Node:  
 def \_\_init\_\_(self, v=int()):  
 self.value = v  
 self.left = None  
 self.right = None  
 self.height\_left = 0  
 self.height\_right = 0  
 self.parent = None  
  
  
class Tree:  
 root = None  
  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, value):  
 if not self.find(value):  
 time = Node(value)  
 if self.root is None:  
 self.root = time  
 else:  
 index = self.root  
 flag = True  
 while flag:  
 if time.value > index.value and index.right is not None:  
 index = index.right  
 elif time.value < index.value and index.left is not None:  
 index = index.left  
 else:  
 flag = False  
 if time.value > index.value:  
 time.parent = index  
 index.right = time  
 else:  
 time.parent = index  
 index.left = time  
 while index is not None:  
 self.root = index  
 index = index.parent  
 return time  
  
 def delete(self, value):  
 self.delete\_top(self.find(value))  
  
 def delete\_top(self, time):  
 if time:  
 if max(time.height\_right, time.height\_left) == 0:  
 if time.parent is None:  
 self.root = None  
 else:  
 index = time.parent  
 if time.parent.left == time:  
 time.parent.left = None  
 else:  
 time.parent.right = None  
 while index is not None:  
 self.root = index  
 index = index.parent  
 else:  
 if time.height\_right > time.height\_left:  
 time.value = time.right.value  
 self.delete\_top(time.right)  
 else:  
 time.value = time.left.value  
 self.delete\_top(time.left)  
  
 def find(self, value):  
 index = self.root  
 flag = True  
 while flag and index is not None:  
 if value > index.value:  
 index = index.right  
 elif value < index.value:  
 index = index.left  
 else:  
 flag = False  
 if index is None:  
 return None  
 else:  
 return index  
  
 def exists(self, value):  
 if self.find(value):  
 return 'true'  
 return 'false'  
  
 def next(self, value):  
 time = self.next\_time(value, self.root)  
 if time == 10 \*\* 10:  
 return 'none'  
 else:  
 return time  
  
 def next\_time(self, value, link):  
 if link is not None:  
 min\_l = self.next\_time(value, link.left)  
 if min\_l <= value:  
 min\_l = 10 \*\* 10  
 min\_r = self.next\_time(value, link.right)  
 if min\_r <= value:  
 min\_r = 10 \*\* 10  
 return min(min\_l, min\_r, link.value if link.value > value else 10 \*\* 10)  
 else:  
 return 10 \*\* 10  
  
 def prev(self, value):  
 time = self.prev\_time(value, self.root)  
 if time == -10 \*\* 11:  
 return 'none'  
 else:  
 return time  
  
 def prev\_time(self, value, link):  
 if link is not None:  
 min\_l = self.prev\_time(value, link.left)  
 if min\_l >= value:  
 min\_l = -10 \*\* 11  
 min\_r = self.prev\_time(value, link.right)  
 if min\_r >= value:  
 min\_r = -10 \*\* 11  
 return max(min\_l, min\_r, link.value if link.value < value else -10 \*\* 11)  
 else:  
 return -10 \*\* 11  
  
  
def main(f):  
 tree = Tree()  
 n = f.readlines()  
 for s in n:  
 s = s.split()  
 if s[0] == 'insert':  
 tree.insert(int(s[1]))  
 elif s[0] == 'exists':  
 print(tree.exists(int(s[1])))  
 elif s[0] == 'prev':  
 print(tree.prev(int(s[1])))  
 elif s[0] == 'next':  
 print(tree.next(int(s[1])))  
 else:  
 tree.delete(int(s[1]))  
  
  
main(open('input.txt'))

Для выполнения задания было создано простое двоичное дерево поиска. Создан класс дерева, в котором расписаны все нужные операции такие как insert, delete, find, prev & next.

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

## Задача №6 Опознание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла]

Нужно проверить, правильно ли реализована структура данных бинарного дерева поиска.

class Node:  
 def \_\_init\_\_(self, key):  
 self.left = None  
 self.right = None  
 self.key = key  
  
  
def check(root):  
 if root is None:  
 return None  
 que = [(root, -1000000000, 10000000000)]  
 while len(que) > 0:  
 line = que.pop(0)  
 isnode = line[0]  
 if isnode.left is not None:  
 if (isnode.left.key >= isnode.key) or (isnode.left.key >= line[2]) or (isnode.left.key <= line[1]):  
 return False  
 que.append((isnode.left, line[1], isnode.key))  
 if isnode.right is not None:  
 if (isnode.right.key <= isnode.key) or (isnode.right.key >= line[2]) or (isnode.right.key <= line[1]):  
 return False  
 que.append((isnode.right, isnode.key, line[2]))  
 return True  
  
  
inp = open('input.txt')  
n = int(inp.readline())  
lines = [inp.readline() for \_ in range(n)]  
nodes = []  
K = []  
L = []  
R = []  
  
for i in range(n):  
 Ki, Li, Ri = lines[i].split()  
 node = Node(int(Ki))  
 nodes.append(node)  
 K.append(int(Ki))  
 L.append(int(Li))  
 R.append(int(Ri))  
  
for i in range(n):  
 left = L[i]  
 right = R[i]  
 if left != -1:  
 nodes[i].left = nodes[left]  
 if right != -1:  
 nodes[i].right = nodes[right]  
inp.close()  
  
result = 0  
if nodes:  
 result = check(nodes[0])  
else:  
 result = True  
  
out = open('output.txt', 'w')  
if result:  
 out.write("CORRECT")  
else:  
 out.write("INCORRECT")  
out.close()

Реализуем алгоритм проверяющий структуру данный бинарного дерева поиска.

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

## Задача №7 Опознание двоичного дерева поиска [10 s, 512 Mb, 1.5 балла]

Нужно проверить, правильно ли реализована структура данных бинарного дерева поиска. (Только усложненная версия)

class Node:  
 def \_\_init\_\_(self, key):  
 self.left = None  
 self.right = None  
 self.key = key  
  
  
def check(root):  
 if root is None:  
 return None  
 que = [(root, -1000000000, 10000000000)]  
 while len(que) > 0:  
 line = que.pop(0)  
 isnode = line[0]  
 if isnode.left is not None:  
 if (isnode.left.key >= isnode.key) or (isnode.left.key >= line[2]) or (isnode.left.key <= line[1]):  
 return False  
 que.append((isnode.left, line[1], isnode.key))  
 if isnode.right is not None:  
 if (isnode.right.key < isnode.key) or (isnode.right.key > line[2]) or (isnode.right.key <= line[1]):  
 return False  
 que.append((isnode.right, isnode.key, line[2]))  
 return True  
  
  
filein = open('input.txt')  
n = int(filein.readline())  
lines = [filein.readline() for \_ in range(n)]  
nodes = []  
K = []  
L = []  
R = []  
  
for i in range(n):  
 Ki, Li, Ri = lines[i].split()  
 node = Node(int(Ki))  
 nodes.append(node)  
 K.append(int(Ki))  
 L.append(int(Li))  
 R.append(int(Ri))  
  
for i in range(n):  
 left = L[i]  
 right = R[i]  
 if left != -1:  
 nodes[i].left = nodes[left]  
 if right != -1:  
 nodes[i].right = nodes[right]  
filein.close()  
  
result = 0  
if nodes:  
 result = check(nodes[0])  
else:  
 result = True  
  
fileout = open('output.txt', 'w')  
if result:  
 fileout.write("CORRECT")  
else:  
 fileout.write("INCORRECT")  
fileout.close()

тот же алгоритм как в 6 задаче , тольку нужно проверять на равные ключи

Изображение выглядит как текст

Автоматически созданное описаниеИзображение выглядит как текст

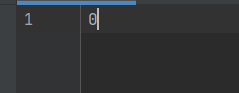
Автоматически созданное описание

## Задача №10 Проверка корректности [2 s, 256 Mb, 2 балла]

Дано двоичное дерево. Проверьте, выполняется ли для него свойство двоичного дерева поиска

class BNode:  
 def \_\_init\_\_(self, value=0, left=0, right=0):  
 self.value = value  
 self.left = left  
 self.right = right  
  
  
def tree\_input():  
 inp = open("input.txt")  
 n = int(inp.readline())  
  
 if n == 0:  
 return True  
 arr = []  
 for \_ in range(n):  
 inp = list(input().split())  
 arr.append(BNode(int(inp[0]), int(inp[1]) - 1, int(inp[2]) - 1))  
 print(arr)  
 result = binary\_tree\_check(arr, 0, -float('inf'), float('inf'))  
 if result:  
 return True  
 return False  
  
  
def binary\_tree\_check(inp, i, left, right):  
 if i == -1:  
 return True  
 if inp[i].value <= left or inp[i].value >= right:  
 return False  
 check = binary\_tree\_check(inp, inp[i].left, left, inp[i].value) and binary\_tree\_check(  
 inp, inp[i].right, inp[i].value, right)  
 return check  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 res = tree\_input()  
 if res:  
 print('YES')  
 else:  
 print('NO')

Реализуем алгоритм проверки на корректность свойства бинарного двоичного дерева.

## Задача №11 Сбалансированное двоичное дерево поиска [2 s, 512 Mb, 2 балла]

Реализуйте сбалансированное двоичное дерево поиска.

class Node:  
 def \_\_init\_\_(self, v=int()):  
 self.value = v  
 self.left = None  
 self.right = None  
 self.height\_left = 0  
 self.height\_right = 0  
 self.parent = None  
  
  
class Tree:  
 root = None  
  
 def \_\_init\_\_(self):  
 self.root = None  
  
 def insert(self, value):  
 if not self.find(value):  
 time = Node(value)  
 if self.root is None:  
 self.root = time  
 else:  
 index = self.root  
 flag = True  
 while flag:  
 if time.value > index.value and index.right is not None:  
 index = index.right  
 elif time.value < index.value and index.left is not None:  
 index = index.left  
 else:  
 flag = False  
 if time.value > index.value:  
 time.parent = index  
 index.right = time  
 else:  
 time.parent = index  
 index.left = time  
 self.balance\_number(index)  
 while index is not None:  
 self.balance\_number(index)  
 index = self.balance(index)  
 self.root = index  
 index = index.parent  
 return time  
  
 def delete(self, value):  
 self.delete\_vertex(self.find(value))  
  
 def delete\_vertex(self, time):  
 if time:  
 if max(time.height\_right, time.height\_left) == 0:  
 if time.parent is None:  
 self.root = None  
 else:  
 index = time.parent  
 if time.parent.left == time:  
 time.parent.left = None  
 else:  
 time.parent.right = None  
 while index is not None:  
 self.balance\_number(index)  
 index = self.balance(index)  
 self.root = index  
 index = index.parent  
 else:  
 if time.height\_right > time.height\_left:  
 time.value = time.right.value  
 self.delete\_vertex(time.right)  
 else:  
 time.value = time.left.value  
 self.delete\_vertex(time.left)  
  
 def find(self, value):  
 index = self.root  
 flag = True  
 while flag and index is not None:  
 if value > index.value:  
 index = index.right  
 elif value < index.value:  
 index = index.left  
 else:  
 flag = False  
 if index is None:  
 return None  
 else:  
 return index  
  
 # log(n)  
 def exists(self, value):  
 if self.find(value):  
 return 'true'  
 return 'false'  
  
 def next(self, value):  
 time = self.next\_time(value, self.root)  
 if time == 10 \*\* 10:  
 return 'none'  
 else:  
 return time  
  
 def next\_time(self, value, link):  
 if link is not None:  
 min\_l = self.next\_time(value, link.left)  
 if min\_l <= value:  
 min\_l = 10 \*\* 10  
 min\_r = self.next\_time(value, link.right)  
 if min\_r <= value:  
 min\_r = 10 \*\* 10  
 return min(min\_l, min\_r, link.value if link.value > value else 10 \*\* 10)  
 else:  
 return 10 \*\* 10  
  
 def prev(self, value):  
 time = self.prev\_time(value, self.root)  
 if time == -10 \*\* 11:  
 return 'none'  
 else:  
 return time  
  
 def prev\_time(self, value, link):  
 if link is not None:  
 min\_l = self.prev\_time(value, link.left)  
 if min\_l >= value:  
 min\_l = -10 \*\* 11  
 min\_r = self.prev\_time(value, link.right)  
 if min\_r >= value:  
 min\_r = -10 \*\* 11  
 return max(min\_l, min\_r, link.value if link.value < value else -10 \*\* 11)  
 else:  
 return -10 \*\* 11  
  
 def balance\_number(self, vertex):  
 if vertex:  
 vertex.height\_left = 0  
 vertex.height\_right = 0  
 if vertex.left:  
 vertex.height\_left = 1 + max(vertex.left.height\_right, vertex.left.height\_left)  
 if vertex.right:  
 vertex.height\_right = 1 + max(vertex.right.height\_right, vertex.right.height\_left)  
  
 def balance(self, vertex):  
 vertex = self.slt(vertex)  
 vertex = self.srt(vertex)  
 vertex = self.blt(vertex)  
 vertex = self.brt(vertex)  
 return vertex  
  
 def slt(self, vertex):  
 if vertex is not None and vertex.height\_right - vertex.height\_left == 2 and vertex.right.height\_left <= vertex.right.height\_right:  
 time\_par = vertex.parent  
 time\_left = vertex.right.left  
 vertex.right.left = vertex  
 vertex.parent = vertex.right  
 vertex.right = time\_left  
 vertex.parent.parent = time\_par  
 if time\_par is not None:  
 if time\_par.left == vertex:  
 time\_par.left = vertex.parent  
 else:  
 time\_par.right = vertex.parent  
 self.balance\_number(vertex)  
 vertex = vertex.parent  
 self.balance\_number(vertex)  
 return vertex  
  
 def srt(self, vertex):  
 if vertex is not None and vertex.height\_left - vertex.height\_right == 2 and vertex.left.height\_right <= vertex.left.height\_left:  
 time\_par = vertex.parent  
 time\_right = vertex.left.right  
 vertex.left.right = vertex  
 vertex.parent = vertex.left  
 vertex.left = time\_right  
 vertex.parent.parent = time\_par  
 if time\_par is not None:  
 if time\_par.left == vertex:  
 time\_par.left = vertex.parent  
 else:  
 time\_par.left = vertex.parent  
 self.balance\_number(vertex)  
 vertex = vertex.parent  
 self.balance\_number(vertex)  
 return vertex  
  
 def blt(self, vertex):  
 if vertex is not None and vertex.height\_right - vertex.height\_left == 2 and vertex.right.height\_left > vertex.right.height\_right:  
 vertex.right = self.srt(vertex.right)  
 vertex = self.slt(vertex)  
 return vertex  
  
 def brt(self, vertex):  
 if vertex is not None and vertex.height\_left - vertex.height\_right == 2 and vertex.left.height\_right <= vertex.left.height\_left:  
 vertex.left = self.slt(vertex.left)  
 vertex = self.srt(vertex)  
 return vertex  
  
  
def main(f):  
 tree = Tree()  
 n = f.readlines()  
 for s in n:  
 s = s.split()  
 if s[0] == 'insert':  
 tree.insert(int(s[1]))  
 elif s[0] == 'exists':  
 print(tree.exists(int(s[1])))  
 elif s[0] == 'prev':  
 print(tree.prev(int(s[1])))  
 elif s[0] == 'next':  
 print(tree.next(int(s[1])))  
 else:  
 tree.delete(int(s[1]))  
  
  
main(open('input.txt'))

Изображение выглядит как текст

Автоматически созданное описание Изображение выглядит как текст

Автоматически созданное описание

## Задача №12 Вставка в АВЛ-дерево [2 s, 256 Mb, 3 балла]

Вставка в АВЛ-дерево вершины V с ключом X при условии, что такой вершины в этом дереве нет, осуществляется

следующим образом.

class Node:  
 def \_\_init\_\_(self, num):  
 self.key = num  
 self.left = None  
 self.right = None  
 self.height = 1  
  
  
class Tree:  
 def height(self, root):  
 return root.height if root is not None else 0  
  
 def balance\_factor(self, root):  
 return self.height(root.right) - self.height(root.left)  
  
 def fix\_height(self, root):  
 left = self.height(root.left)  
 right = self.height(root.right)  
 root.height = max(left, right) + 1  
  
 def rotateR(self, root):  
 q = root.left  
 root.left = q.right  
 q.right = root  
 self.fix\_height(root)  
 self.fix\_height(q)  
 return q  
  
 def rotateL(self, root):  
 p = root.right  
 root.right = p.left  
 p.left = root  
 self.fix\_height(root)  
 self.fix\_height(p)  
 return p  
  
 def balance(self, root):  
 self.fix\_height(root)  
 if self.balance\_factor(root) == 2:  
 if self.balance\_factor(root.right) < 0:  
 root.right = self.rotateR(root.right)  
 return self.rotateL(root)  
 if self.balance\_factor(root) == -2:  
 if self.balance\_factor(root.left) > 0:  
 root.left = self.rotateL(root.left)  
 return self.rotateR(root)  
 return root  
  
 def insert(self, root, key):  
 if root is None:  
 return Node(key)  
 if key < root.key:  
 root.left = self.insert(root.left, key)  
 else:  
 root.right = self.insert(root.right, key)  
 return self.balance(root)  
  
 def get\_str(self, root):  
 que = []  
 number = 1  
 que.append(root)  
 ans = []  
 while len(que) > 0:  
 node = que.pop(0)  
 line = f"{node.key} "  
 if node.left is not None:  
 number += 1  
 line += f"{number} "  
 que.append(node.left)  
 else:  
 line += "0 "  
  
 if node.right is not None:  
 number += 1  
 line += f"{number}\n"  
 que.append(node.right)  
 else:  
 line += "0\n"  
 ans.append(line)  
 return ans  
  
  
inp = open("input.txt")  
  
n = int(inp.readline())  
lines = [inp.readline() for \_ in range(n)]  
lines.reverse()  
nodes = {}  
  
tree = Tree()  
for i in range(n):  
 Ki, Li, Ri = map(int, lines[i].split())  
 node = Node(int(Ki))  
 nodes[n - i] = node  
 if Li != 0 and Ri != 0:  
 node.left = nodes[Li]  
 node.right = nodes[Ri]  
 tree.fix\_height(node)  
 elif Li != 0:  
 node.left = nodes[Li]  
 tree.fix\_height(node)  
 elif Ri != 0:  
 node.right = nodes[Ri]  
 tree.fix\_height(node)  
 else:  
 node.height = 1  
  
if len(nodes) == 0:  
 nodes[1] = None  
  
key = int(inp.readline())  
root = tree.insert(nodes[1], key)  
  
out = open("output.txt", "w")  
ans = tree.get\_str(root)  
out.write(f"{len(ans)}\n")  
for i in ans:  
 out.write(i)  
  
out.close()  
inp.close()

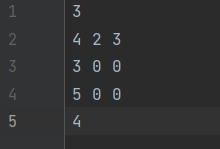
Изображение выглядит как текст

Автоматически созданное описание 

## Задача №15 Удаление из АВЛ-дерева [2 s, 256 Mb, 3 балла]

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

class Node:  
 def \_\_init\_\_(self, num):  
 self.key = num  
 self.left = None  
 self.right = None  
 self.height = 1  
  
  
class Tree:  
 def height(self, root):  
 return root.height if root is not None else 0  
  
 def balance\_factor(self, root):  
 return self.height(root.right) - self.height(root.left)  
  
 def fix\_height(self, root):  
 left = self.height(root.left)  
 right = self.height(root.right)  
 root.height = max(left, right) + 1  
  
 def rotateR(self, root):  
 q = root.left  
 root.left = q.right  
 q.right = root  
 self.fix\_height(root)  
 self.fix\_height(q)  
 return q  
  
 def rotateL(self, root):  
 p = root.right  
 root.right = p.left  
 p.left = root  
 self.fix\_height(root)  
 self.fix\_height(p)  
 return p  
  
 def balance(self, root):  
 self.fix\_height(root)  
 if self.balance\_factor(root) == 2:  
 if self.balance\_factor(root.right) < 0:  
 root.right = self.rotateR(root.right)  
 return self.rotateL(root)  
 if self.balance\_factor(root) == -2:  
 if self.balance\_factor(root.left) > 0:  
 root.left = self.rotateL(root.left)  
 return self.rotateR(root)  
 return root  
  
 def insert(self, root, key):  
 if root is None:  
 return Node(key)  
 if key < root.key:  
 root.left = self.insert(root.left, key)  
 else:  
 root.right = self.insert(root.right, key)  
 return self.balance(root)  
  
 def find\_right(self, root):  
 if root.right is not None:  
 return self.find\_right(root.right)  
 return root  
  
 def find\_right\_and\_delete(self, root):  
 if root.right is not None:  
 if root.right.right is None:  
 root.right = root.right.left if (root.right.left is not None) else None  
 else:  
 root.right = self.find\_right\_and\_delete(root.right)  
 return self.balance(root)  
  
 def remove(self, root, key):  
 if root is None:  
 return None  
 if key < root.key:  
 root.left = self.remove(root.left, key)  
 elif key > root.key:  
 root.right = self.remove(root.right, key)  
 else:  
 if (root.left is None) and (root.right is None):  
 return None  
 elif root.left is None:  
 return root.right  
 else:  
 new\_root = self.find\_right(root.left)  
 root.key = new\_root.key  
 if root.left.key == new\_root.key:  
 root.left = None if (root.left.left is None) else root.left.left  
 else:  
 root.left = self.find\_right\_and\_delete(root.left)  
 return self.balance(root)  
  
 def get\_str(self, root):  
 if root is None:  
 return None  
 que = []  
 number = 1  
 que.append(root)  
 ans = []  
 while len(que) > 0:  
 node = que.pop(0)  
 line = f"{node.key} "  
 if node.left is not None:  
 number += 1  
 line += f"{number} "  
 que.append(node.left)  
 else:  
 line += "0 "  
  
 if node.right is not None:  
 number += 1  
 line += f"{number}\n"  
 que.append(node.right)  
 else:  
 line += "0\n"  
 ans.append(line)  
 return ans  
  
  
inp = open("input.txt")  
  
n = int(inp.readline())  
lines = [inp.readline() for \_ in range(n)]  
lines.reverse()  
nodes = {}  
  
tree = Tree()  
for i in range(n):  
 Ki, Li, Ri = map(int, lines[i].split())  
 node = Node(int(Ki))  
 nodes[n - i] = node  
 if Li != 0 and Ri != 0:  
 node.left = nodes[Li]  
 node.right = nodes[Ri]  
 tree.fix\_height(node)  
 elif Li != 0:  
 node.left = nodes[Li]  
 tree.fix\_height(node)  
 elif Ri != 0:  
 node.right = nodes[Ri]  
 tree.fix\_height(node)  
 else:  
 node.height = 1  
  
if len(nodes) == 0:  
 nodes[1] = None  
  
key = int(inp.readline())  
root = tree.remove(nodes[1], key)  
  
out = open("output.txt", "w")  
ans = tree.get\_str(root)  
if ans is not None:  
 out.write(f"{len(ans)}\n")  
 for i in ans:  
 out.write(i)  
else:  
 out.write("0")  
  
out.close()  
inp.close()

 Изображение выглядит как текст

Автоматически созданное описание