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: BST

Вариант 2

Выполнила:
Бочкарева Е.А.
К3144

Проверила:
Артамонова В.Е.

Санкт-Петербург
2024 г.

Содержание отчета

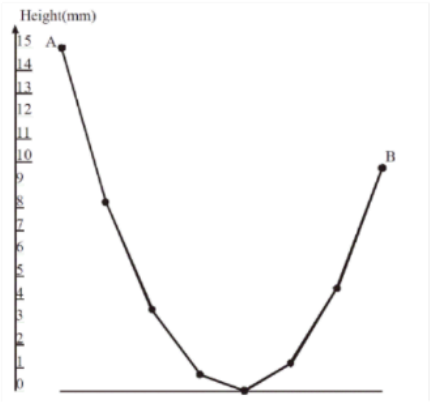
Задачи по варианту

Задача №2 Гирлянда

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 ($h_i = \frac{h_{i-1} + h_{i+1}}{2} - 1$ для $1 < i < N$).

Требуется найти 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.

Подсказка: для решения этой задачи можно использовать двоичный поиск.



- **Формат ввода / входного файла (input.txt).** В первой строке входного файла содержится два числа n и A.
- **Ограничения на входные данные.** $3 \le n \le 1000$, n – целое, $10 \le A \le 1000$, A – вещественное и дано не более чем с тремя знаками после десятичной точки.
- **Формат вывода / выходного файла (output.txt).** Выведите одно вещественное число B – минимальную высоту второго конца. Ваш ответ будет засчитан, если он будет отличаться от правильного не более, чем на 10^{-6}.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.
- Примеры:

input.txt	output.txt
8 15	9.75
692 532.81	446113.34434782615

Листинг кода:

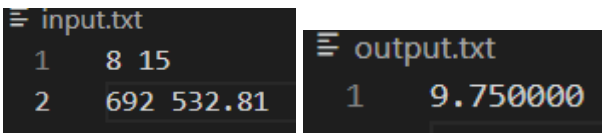
```
def main():
    with open("input.txt") as fin:
        host_line = fin.readline().split()
        n, A = int(host_line[0]), float(host_line[1])
        result, error = 10 ** 9, 0.1 ** 10
        heights = [0] * n
        heights[0] = A
        left, right = 0, heights[0]

        while abs(right - left) > error:
```

```
            heights[1] = (left + right) / 2
            heights[-1] = 0
            is_up = False

            for i in range(2, n):
                heights[i] = 2 * heights[i - 1] - heights[i - 2] + 2
                if heights[i] <= error:
                    is_up = True
                    break

            if heights[-1] > error:
                result = min(result, heights[-1])
            if is_up:
                left = heights[1]
            else:
                right = heights[1]

        return result

if __name__ == "__main__":
    with open("output.txt", "w") as fout:
        print("%.6f" % main(), file=fout)
```

Пояснение к решению:

Если H[i] = (H[i-1]+H[i+1]) / 2 – 1, то мы можем прийти к уравнению H[i+1] = 2H[i] – H[i-1]. Задача требует найти значение высоты второй лампочки. Используем двоичный поиск, чтобы установить, сколько разрешенных областей. Установим переменную error, которая является погрешностью, указывающей на ту малую область, в которой значения равны.

Результат работы на примерах из текста к задаче:



Вывод по задаче:

В этой задаче я попробовала применение бинарного поиска.

Задача №1 Обход двоичного дерева

В этой задаче вы реализуете три основных способа обхода двоичного дерева «в глубину»: центрированный (in-order), прямой (pre-order) и обратный (post-order). Очень полезно попрактиковаться в их реализации, чтобы лучше понять бинарные деревья поиска.

Вам дано корневое двоичное дерево. Выведите центрированный (in-order), прямой (pre-order) и обратный (post-order) обходы в глубину.

- **Формат ввода: стандартный ввод или input.txt.** В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до $n-1$. Узел 0 является корнем.

  Следующие n строк содержат информацию об узлах $0, 1, ..., n-1$ по порядку. Каждая из этих строк содержит три целых числа K_i, L_i и R_i. K_i – ключ i-го узла, L_i - индекс левого ребенка i-го узла, а R_i - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа L_i или R_i (или оба) будут равны -1.

- **Ограничения на входные данные.** $1 \le n \le 10^5, 0 \le K_i \le 10^9, -1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла.

- **Формат вывода / выходного файла (output.txt).** Выведите три строки. Первая строка должна содержать ключи узлов при центрированном обходе дерева (in-order). Вторая строка должна содержать ключи узлов при прямом обходе дерева (pre-order). Третья строка должна содержать ключи узлов при обратном обходе дерева (post-order).

- Ограничение по времени. 5 сек.

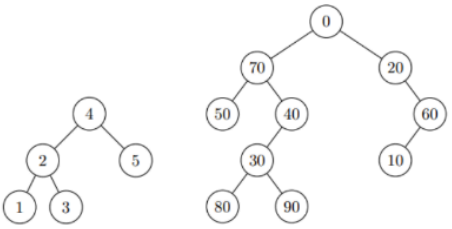
- Ограничение по памяти. 512 мб.

- Примеры:

input	output.txt
5 4 1 2 2 3 4 5 -1 -1 1 -1 -1 3 -1 -1	1 2 3 4 5 4 2 1 3 5 1 3 2 5 4

input	output.txt
10 0 7 2 10 -1 -1 20 -1 6 30 8 9 40 3 -1 50 -1 -1 60 1 -1 70 5 4 80 -1 -1 90 -1 -1	50 70 80 30 90 40 0 20 10 60 0 70 50 40 30 80 90 20 60 10 50 80 90 30 40 70 10 60 20 0

- Иллюстрации к обоим примерам:



- Что делать. Реализовать алгоритмы обхода из лекции. Обратите внимание, что в этой задаче дерево может быть очень глубоким, поэтому будьте осторожны, избегайте проблем с переполнением стека, если используете рекурсию, и обязательно протестируйте свое решение на дереве максимально возможной высоты.

Листинг кода:

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

def create_bst():
    with open("input.txt") as fin:
        n = int(fin.readline())
```

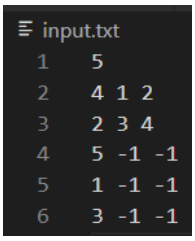
```
        lines = [fin.readline().strip() for _ in range(n)]
        nodes, k, l, r = [], [], [], []

        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]

    return nodes[0]

def in_order(tree):
    if tree:
        in_order(tree.left)
        print(tree.key, end=" ", file=fout)
        in_order(tree.right)

def pre_order(tree):
    if tree:
        print(tree.key, end=" ", file=fout)
        pre_order(tree.left)
        pre_order(tree.right)

def post_order(tree):
    if tree:
        post_order(tree.left)
        post_order(tree.right)
        print(tree.key, end=" ", file=fout)

def main():
    tree = create_bst()
    in_order(tree)
    print(file=fout)
```

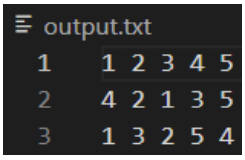
```
    pre_order(tree)
    print(file=fout)
    post_order(tree)

if __name__ == "__main__":
    with open("output.txt", "w") as fout:
        main()
```

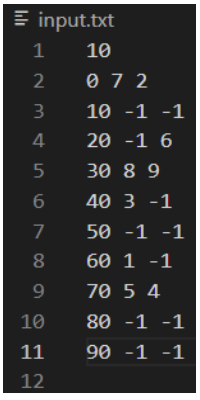
Пояснение к решению:

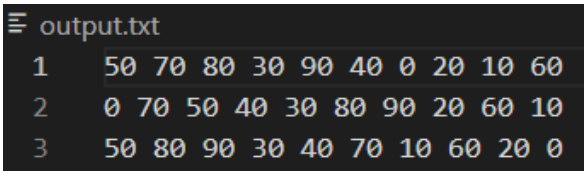
Программа создает бинарное дерево поиска с помощью данных из файла input.txt. Для этого создается класс, который содержит ключ и указатели на левую и правую вершины. Затем создается функция create_bst(), которая считывает данные из файла input.txt, создает узлы и присваивает им соответствующие левые и правые указатели. Затем создаются 3 функции: in_order(), pre_order() и post_order(), которые выполняют обходы для BST. Результаты обхода записываются в файл output.txt.

Результат работы на примерах из текста к задаче:









Вывод по задаче:

Научилась делать обход двоичного дерева.

Задача №7 Опознание двоичного дерева поиска (усложненная версия)

Эта задача отличается от предыдущей тем, что двоичное дерева поиска может содержать равные ключи.

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 поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева **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.

- **Формат ввода / входного файла (input.txt).** В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до $n-1$. Узел 0 является корнем.

  Следующие n строк содержат информацию об узлах $0, 1, ..., n-1$ по порядку. Каждая из этих строк содержит три целых числа K_i, L_i и R_i. K_i – ключ i-го узла, L_i - индекс левого ребенка i-го узла, а R_i - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа L_i или R_i (или оба) будут равны -1.

- **Ограничения на входные данные.** $0 \le n \le 10^5$, $-2^{31} \le K_i \le 2^{31}-1$, $-1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла. Обратите внимание, что минимальное и максимальное возможные значения 32-битного целочисленного типа могут быть ключами в дереве.

- **Формат вывода / выходного файла (output.txt).** Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).

- Ограничение по времени. 10 сек.

- Ограничение по памяти. 512 мб.

- Примеры:

input.txt	output.txt
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT

input.txt	output.txt
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT

input.txt	output.txt
3 2 1 2 1 -1 -1 2 -1 -1	CORRECT

input.txt	output.txt
3 2 1 2 2 -1 -1 3 -1 -1	INCORRECT

input.txt	output.txt
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT

input.txt	output.txt
7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT

input.txt	output.txt
1 2147483647 -1 -1	CORRECT

Листинг кода:

```
class TreeNode(object):
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None

    def check(self, root):
        if not root:
            return True
```

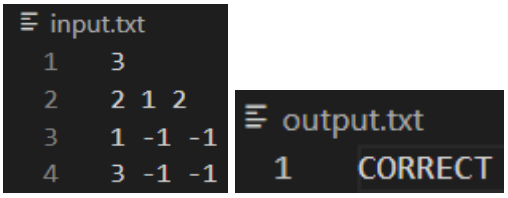
```
        level = [[root, -float("inf"), float("inf")]]
        while level:
            next_level = []
            for element in level:
                node, min_val, max_val = element
                if min_val <= node.val < max_val:
                    if node.left:
                        next_level.append([node.left, min_val,
node.val])
                    if node.right:
                        next_level.append([node.right, node.val,
max_val])
                else:
                    return False
            level = next_level
        return True

def main():
    with open("input.txt") as f:
        n = int(f.readline())
        a = []
        mr = -999999999999
        ml = 99999999999
        for i in range(n):
            q = list(map(int, f.readline().split()))
            if q[0] > mr:
                mr = q[0]
            if q[0] < ml:
                ml = q[0]
            a.append(q)

        for i in range(n):
            a[i][0] = TreeNode(a[i][0])

        for i in range(n):
            if a[i][1] != -1:
                a[i][0].left = a[a[i][1]][0]
            if a[i][2] != -1:
                a[i][0].right = a[a[i][2]][0]

        er = True
        if n > 0:
            er = a[0][0].check(a[0][0])
```

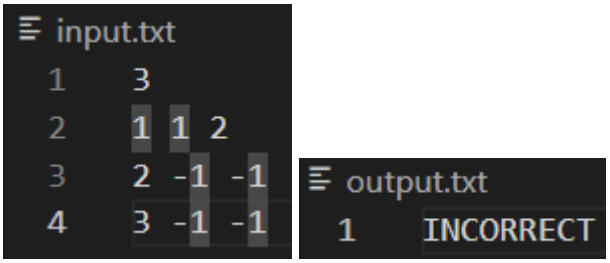
```
    with open("output.txt", "w") as fout:
        if er:
            fout.write("CORRECT")
        else:
            fout.write("INCORRECT")

if __name__ == "__main__":
    main()
```

Пояснение к решению:

Для реализации дерева создается класс TreeNode. В нем же создается новая функция check. Мы отслеживаем, что значение узла лежит между верхним и нижним пределами. Если мы идем на левом поддереве, то мы будем использовать значение текущего узла в качестве нового верхнего предела и сохранять исходный нижний предел. Если мы на правом поддереве, то мы будем использовать значение текущего узла в качестве нижнего предела и сохранять исходный верхний предел.

Результат работы на некоторых примерах из текста к задаче:





Вывод по задаче:

Опознала двоичное дерево поиска.

Задача №16 K-й максимум

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

- **Формат ввода / входного файла (input.txt).** Первая строка входного файла содержит натуральное число n – количество команд. Последующие n строк содержат по одной команде каждая. Команда записывается в виде двух чисел c_i и k_i – тип и аргумент команды соответственно. Поддерживаемые команды:
  - +1 (или просто 1): Добавить элемент с ключом k_i.
  - 0 : Найти и вывести k_i-й максимум.
  - -1 : Удалить элемент с ключом k_i.

  Гарантируется, что в процессе работы в структуре не требуется хранить элементы с равными ключами или удалять несуществующие элементы. Также гарантируется, что при запросе ki-го мак- симума, он существует.
- **Ограничения на входные данные.** $n \leq 100000$, $|k_i| \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** Для каждой команды нулевого типа в выходной файл должна быть выведена строка, содержащая единственное число – k_i-й максимум.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 512 мб.
- Пример:

input.txt	output.txt
11	7
+1 5	5
+1 3	3
+1 7	10
0 1	7
0 2	3
0 3	
-1 5	
+1 10	
0 1	
0 2	
0 3	

Листинг кода:

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

    def __repr__(self):
        return self.key


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

    def insert(self, key):
        if self.root is None:
            self.root = Node(key)
```

```
        else:
            self.insert_recursive(self.root, key)

    def insert_recursive(self, node, key):
        if key < node.key:
            if node.left is None:
                node.left = Node(key)
            else:
                self.insert_recursive(node.left, key)
        elif key > node.key:
            if node.right is None:
                node.right = Node(key)
            else:
                self.insert_recursive(node.right, key)

    def delete(self, key):
        self.root = self.delete_recursive(self.root, key)

    def delete_recursive(self, node, key):
        if node is None:
            return None
        if key < node.key:
            node.left = self.delete_recursive(node.left, key)
        elif key > node.key:
            node.right = self.delete_recursive(node.right, key)
        else:
            if node.left is None:
                return node.right
            elif node.right is None:
                return node.left
            else:
                successor = self.min(node.right)
                node.key = successor.key
                node.right = self.delete_recursive(node.right,
successor.key)
        return node

    def min(self, node):
        curr = node
        while curr.left is not None:
            curr = curr.left
        return curr
```

```
    def find_k_max(self, k):
        curr = self.root
        k_largest = None
        count = 0  # счетчик посещенных узлов
        while curr is not None:
            if curr.right is None:
                count += 1
                if count == k:
                    k_largest = curr
                curr = curr.left
            else:
                succ = curr.right
                while succ.left is not None and succ.left != curr:
                    succ = succ.left
                if succ.left is None:
                    succ.left = curr
                    curr = curr.right
                else:
                    succ.left = None
                    count += 1
                    if count == k:
                        k_largest = curr
                    curr = curr.left
        return k_largest.__repr__()


def main():
    with open("input.txt", "r") as file:
        n = int(file.readline().strip())
        operations = [list(map(int, file.readline().strip().split()))
for _ in range(n)]

    kmax = KMaxStructure()
    result = []

    for op in operations:
        if op[0] == 1:
            kmax.insert(op[1])
        elif op[0] == 0:
            result.append(str(kmax.find_k_max(op[1])))
        elif op[0] == -1:
            kmax.delete(op[1])
```

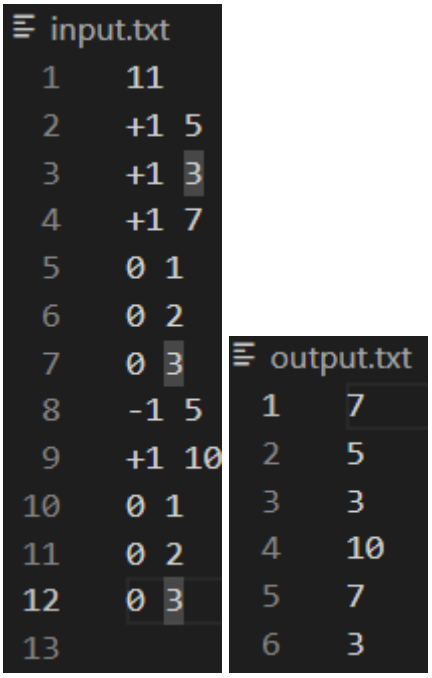
```
    with open("output.txt", "w") as file:
        file.write("\n".join(result))

if __name__ == "__main__":
    main()
```

Пояснение к решению:

Используем двоичное дерево поиска. Реализуется класс Node для представления узла дерева и класс KMaxStructure для представления самого дерева. Класс KMaxStructure включает в себя методы для добавления и удаления ключа, а также поиска и вывода k-го максимального элемента: insert, delete и find_k_max.

Функции insert и delete реализованы обычным способом для простого двоичного дерева поиска.

Результат работы на примерах из текста к задаче:



Вывод по задаче:

В этой задаче я научилась находить максимум, добавлять и убирать элементы.

Дополнительные задачи

Задача №3 Простейшее BST

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.

- **Формат ввода / входного файла (input.txt).** В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.
- Случайные данные! Не нужно ничего специально балансировать.
- **Ограничения на входные данные.** $1 \leq x \leq 10^9, 1 \leq N \leq 300000$
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса вида «> x» выведите в отдельной строке ответ.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
+ 1	3
+ 3	3
+ 3	0
> 1	2
> 2	
> 3	
+ 2	
> 1	

Листинг кода:

```
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
```

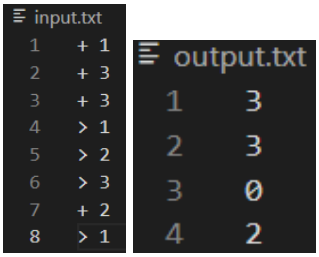
```
            elif key > node.key:
                node = node.right
            else:
                return

        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 find_min(self, x):
        if self.root is None:
            return 0

        path = []
        node = self.root

        while True:
            path.append(node)
            if x > node.key:
                if node.right is None:
                    break
                node = node.right
            elif x < node.key:
                if node.left is None:
                    break
                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(path) - 1, -1, -1):
            if path[i].key > x:
```

```
                return path[i].key

        return 0

def main():
    tree = Tree()
    with open("input.txt") as fin:
        line = fin.readline()
        with open("output.txt", "w") as fout:
            while line:
                items = line.split()
                if items[0] == "+":
                    tree.insert(int(items[1]))
                else:
                    print(tree.find_min(int(items[1])), file=fout)
                line = fin.readline()

if __name__ == "__main__":
    main()
```

Пояснение к решению:

Функция ищет минимальное значение, которое больше заданного числа x в дереве. Для этого мы проходимся по дереву, сравнивая x с ключом узла. Если x больше ключа узла – идем в правое поддерево, если меньше – проверяем, существует ли у узла левое поддерево. В противном случае, мы идём вниз, проверяя, что ключ узла больше x. В противном случае, мы идем вниз, пока не дойдем до узла, который является корневым, т. е. не имеет “подузлов”.

Результат работы на примерах из текста к задаче:



Вывод по задаче:

В данной задаче я научилась писать простейшее BST.

Задача №4 Простейший неявный ключ

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.

- **Формат ввода / входного файла (input.txt).** В каждой строке содержится один запрос. Все x - целые числа, количество запросов N не указано в начале, не более 300 000. Гарантируется, что все x выбраны равномерным распределением.
- Случайные данные! Не нужно ничего специально балансировать.
- **Ограничения на входные данные.** $1 \le x \le 10^9$, $1 \le N \le 300000$, в запросах «? k», число k от 1 до количества элементов в дереве.
- **Формат вывода / выходного файла (output.txt).** Для каждого запроса вида «? k» выведите в отдельной строке ответ.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
+ 1	1
+ 4	3
+ 3	4
+ 3	3
? 1	
? 2	
? 3	
+ 2	
? 3	

Листинг кода:

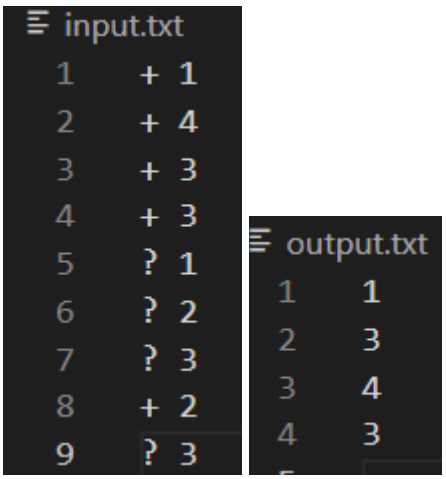
```
def binary_search(a, left, right, x):
    while left <= right:
        mid = left + (right - left) // 2
        if a[mid] == x:
            return None
        if a[mid] < x:
            right = mid - 1
        else:
            left = mid + 1
    return right

class BST:
    def __init__(self):
        self.a = []

    def insert(self, val):
        index = binary_search(self.a, 0, len(self.a) - 1, val)
        if index is not None:
```

```
            self.a = self.a[: index + 1] + [val] + self.a[index + 1:]

def main():
    bst = BST()
    result = ""
    with open("input.txt") as fin:
        for i in fin.readlines():
            cmd = i.split()
            v, cmd = cmd[1], cmd[0]
            if cmd == "+" and int(v):
                bst.insert(int(v))
            if cmd == "?":
                result += str(bst.a[-int(v)]) + "\n"
    with open("output.txt", "w") as fout:
        print(result, file=fout)

if __name__ == "__main__":
    main()
```

Пояснение к решению:

С помощью бинарного поиска добавляем значения в массив.

Результат на примерах из задачи:



Вывод по задаче:

В данной задаче я научилась писать BST по неявному ключу и отвечать им на запросы из условия.

Задача №6 Опознание двоичного дерева поиска

В этой задаче вы собираетесь проверить, правильно ли реализована структура данных бинарного дерева поиска. Другими словами, вы хотите убедиться, что вы можете находить целые числа в этом двоичном дереве, используя бинарный поиск по дереву, и вы всегда получите правильный результат: если целое число есть в дереве, вы его найдете, иначе – нет.

Вам дано двоичное дерево с ключами - целыми числами. Вам нужно проверить, является ли это правильным двоичным деревом поиска. Для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева больше ключа вершины V.

Другими словами, узлы с меньшими ключами находятся слева, а узлы с большими ключами – справа. Вам необходимо проверить, удовлетворяет ли данная структура двоичного дерева этому условию. Вам гарантируется, что входные данные содержат допустимое двоичное дерево. То есть это дерево, и каждый узел имеет не более двух ребенков.

- **Формат ввода / входного файла (input.txt).** В первой строке входного файла содержится количество узлов n. Узлы дерева пронумерованы от 0 до $n-1$. Узел 0 является корнем.

  Следующие n строк содержат информацию об узлах $0, 1, ..., n-1$ по порядку. Каждая из этих строк содержит три целых числа K_i, L_i и R_i. K_i – ключ i-го узла, L_i - индекс левого ребенка i-го узла, а R_i - индекс правого ребенка i-го узла. Если у i-го узла нет левого или правого ребенка (или обоих), соответствующие числа L_i или R_i (или оба) будут равны -1.

- **Ограничения на входные данные.** $0 \le n \le 10^5$, $-2^{31} \le K_i \le 2^{31}-1$, $-1 \le L_i, R_i \le n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \ne -1$ и $R_i \ne -1$, то $L_i \ne R_i$. Кроме того, узел не может быть ребенком двух разных узлов. Кроме того, каждый узел является потомком корневого узла.

  Все ключи во входных данных различны.

- **Формат вывода / выходного файла (output.txt).** Если заданное двоичное дерево является правильным двоичным деревом поиска, выведите одно слово «CORRECT» (без кавычек). В противном случае выведите одно слово «INCORRECT» (без кавычек).

- Ограничение по времени. 10 сек.

- Ограничение по памяти. 512 мб.

- Примеры:

input.txt	output.txt
3 2 1 2 1 -1 -1 3 -1 -1	CORRECT

input.txt	output.txt
3 1 1 2 2 -1 -1 3 -1 -1	INCORRECT

input.txt	output.txt
0	CORRECT

input.txt	output.txt
5 1 -1 1 2 -1 2 3 -1 3 4 -1 4 5 -1 -1	CORRECT

input.txt	output.txt
7 4 1 2 2 3 4 6 5 6 1 -1 -1 3 -1 -1 5 -1 -1 7 -1 -1	CORRECT

input.txt	output.txt
4 4 1 -1 2 2 3 1 -1 -1 5 -1 -1	INCORRECT

Листинг кода:

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

def check(root):
    if root is None:
```

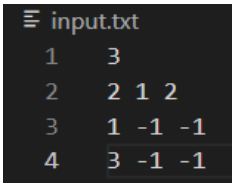
```
        return None

    queue = [(root, -10**9, 10**9)]

    while len(queue) > 0:
        node, min_val, max_val = queue.pop(0)

        if node.left is not None:
            if node.left.key >= node.key or node.left.key >= max_val or
node.left.key <= min_val:
                return False
            queue.append((node.left, min_val, node.key))

        if node.right is not None:
            if node.right.key <= node.key or node.right.key >= max_val
or node.right.key <= min_val:
                return False
            queue.append((node.right, node.key, max_val))

    return True

def main():
    with open("input.txt") as fin:
        n = int(fin.readline())
        lines = [fin.readline().strip() for _ in range(n)]

        nodes, k, l, r = [], [], [], []
        for line in lines:
            ki, li, ri = map(int, line.split())
            nodes.append(Node(ki))
            k.append(ki)
            l.append(li)
            r.append(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]

    with open("output.txt", "w") as fout:
```

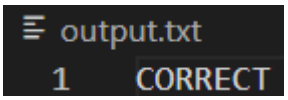
```
        if nodes:
            result = check(nodes[0])
        else:
            result = True

        if result:
            fout.write("CORRECT")
        else:
            fout.write("INCORRECT")

if __name__ == "__main__":
    main()
```

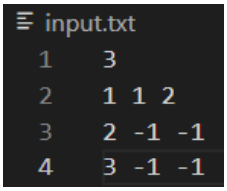
Пояснение к решению:

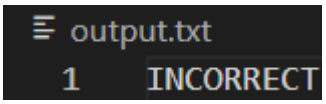
Строим бинарное дерево и заносим в очередь его корень. Затем, спускаясь по дереву, извлекаем родителей, хранящихся в виде кортежа, содержащих своих потомков, и заносим их в очередь. Если один из детей будет выходить за рамки установленных минимального или максимального значения, то строить такое дерево уже невозможно.

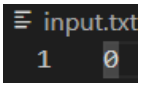
Результат на примерах из задачи:

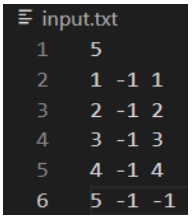


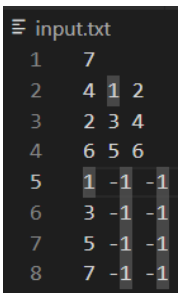


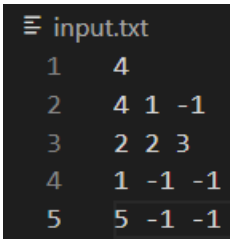


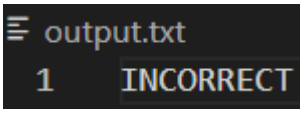












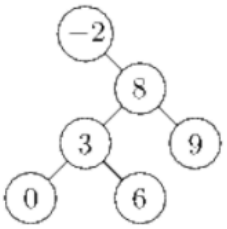
Вывод по задаче:

Двоичные деревья поиска являются очень эффективным способом хранения, организации, поиска и обмена данными. В этой задаче я вновь рассмотрела возможности этой структуры данных.

Задача №8 Высота дерева возвращается

Высотой дерева называется максимальное число вершин дерева в цепочке, начинающейся в корне дерева, заканчивающейся в одном из его листьев, и не содержащей никакую вершину дважды.

Так, высота дерева, состоящего из единственной вершины, равна единице. Высота пустого дерева равна нулю. Высота дерева, изображенного на рисунке, равна четырем.



Дано **двоичное дерево поиска**. В вершинах этого дерева записаны ключи – целые числа, по модулю не превышающие 10^9. Для каждой вершины дерева V выполняется следующее условие:

- все ключи вершин из левого поддерева меньше ключа вершины V;
- все ключи вершин из правого поддерева больше ключа вершины V.

Найдите высоту данного дерева.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева. В первой строке файла находится число N – число вершин в дереве. В последующих N строках файла находятся описания вершин дерева. В $(i + 1)$-ой строке файла $(1 \le i \le N)$ находится описание i-ой вершины, состоящее из трех чисел K_i, L_i, R_i, разделенных пробелами – ключа K_i в i-ой вершине, номера левого L_i ребенка i-ой вершины $(i < L_i \le N$ или $L_i = 0$, если левого ребенка нет) и номера правого R_i ребенка i-ой вершины $(i < R_i \le N$ или $R_i = 0$, если правого ребенка нет).
- **Ограничения на входные данные.** $0 \le N \le 2 \cdot 10^5, |K_i| \le 10^9$. Все ключи различны. Гарантируется, что данное дерево является деревом поиска.
- **Формат вывода / выходного файла (output.txt).** Выведите одно целое число – высоту дерева.
- **Ограничение по времени. 2 сек.**
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
6 -2 0 2 8 4 3 9 0 0 3 6 5 6 0 0 0 0 0	4

Листинг кода:

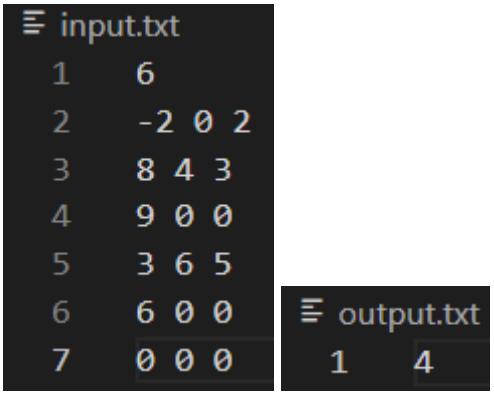
```
def main():
    with open("input.txt") as fin:
        n = int(fin.readline())
        woods = []
        deeps = [0] * (n + 1)

        for _ in range(n):
            value, left, right = map(int, fin.readline().split())
            woods.append((left, right))
```

```
        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

    with open("output.txt", "w") as fout:
        if n > 0:
            print(deeps[1], file=fout)
        else:
            fout.write("0")

if __name__ == "__main__":
    main()
```

Пояснение к решению:

Для вычисления высоты дерева будем разбивать его на поддеревья. Начнем с листьев, у которых высота равна 1. Для остальных узлов будем считать, что их высота равна 1 + максимуму среди дочерних узлов.

Результат на примерах из задачи:



Вывод по задаче:

Научилась писать алгоритм для вычисления высоты дерева.