Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики

Алгоритмы и структуры данных

Лабораторная работа №2_2

Двоичные деревья поиска

Выполнил:

Бараканов Жаргал Мырзабекович

Факультет ИКТ

Группа К3121

Преподаватель:

Харьковская Татьяна Александровна

Санкт-Петербург

2022

Для удобства понимая функций и алгоритмов, применяемых в ходе данной лабораторной работы, разделим задачи по типу ввода данных:

I. Выполнение команд, записанных в исходном файле;
II. Ввод всего дерева сразу и дальнейшая работа с ним.

I. Выполнение команд, записанных в исходном файле

Задание 3.

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.** $1 \le x \le 10^9$, $1 \le N \le 300000$
- **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.

Решение:

Для работы с деревьями были созданы классы «node» и «binTree». Первый отвечает за хранение ключа, левого и правого потомков, а также родителя соотвествующего узла. Во втором хранится корень дерева, словарь «nodes», роль которого будет объяснена ниже, а функции, которык будут необходимы для работы с деревом.

```
class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

И первой из этих функций будет функция вставки. Созданный в конструкторе класса словарь нужен для хранения всех узлов дерева. Узел будет помещён в дерево, только если в дереве нет узла с таким ключом. Далее происходит поиск места, куда будет вставлен новый узел, в соответствии с правилами двоичного дерева поиска, и соответственно вставка нового узла с присущими ему атрибутами.

```
    def insert(self, key):
        if key not in self.nodes:
            self.nodes[key] = node()
            self.nodes[key].key = key
            if not tree.root:
                tree.root = self.nodes[key]
            else:
                root = tree.root
                while True:
                    if key < root.key:
                        if not root.left:
                            root.left = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.left
                    else:
                        if not root.right:
                            root.right = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.right
```

Следующей на очереди функцией является функция нахождения наименьшего ключа, большего, чем поданное в программу число. Поиск происходит с помощью словаря с узлами данного дерева. Далее если такой ключ был найден, то он выводится на экран, в противном случае, выводится 0.

```
    def findMinMore(self, key):
        min = float('inf')
        for j in self.nodes:
            if self.nodes[j].key > key and key < min:
                min = self.nodes[j].key
        if min == float('inf'):
            print(0)
        else:
            print(min)
```

В основном коде непосредственно происходит считывание команд из файла. В зависимости от введенной команды выполняется соответствующая функция.

```
tree = binTree()
with open('input.txt') as f:
    while True:
        command = f.readline()
        if command == '':
            break
        command = list(map(str, command.split()))
        key = int(command[1])
        command = command[0]
        if command == '+':
            tree.insert(key)
        else:
            tree.findMinMore(key)
```

Задание 4.

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.

Решение:

Классы и их конструкторы остались те же самые, только в классе «node» добавился атрибут «order», отвечающий за порядок ключа в множестве ключей дерева.

```
class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.order = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

Функция вставки ключа осталась почти той же самой. Было добавлено лишь присвоение атрибута «order», равного 1. Также в конце происходит проход по словарю «nodes». Если ключ очередного узла больше текущего, то порядок очередного узла увеличивается на 1, и наоборот.

```
    def insert(self, key):
        if key not in self.nodes:
            self.nodes[key] = node()
            self.nodes[key].key = key
            self.nodes[key].order = 1
            if not tree.root:
                tree.root = self.nodes[key]
            else:
                root = tree.root
                while True:
                    if key < root.key:
                        if not root.left:
                            root.left = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.left
                    else:
                        if not root.right:
                            root.right = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.right
                for i in self.nodes:
                    if self.nodes[i].key > self.nodes[key].key:
                        self.nodes[i].order += 1
                    elif i != key:
                        self.nodes[key].order += 1
```

В основном коде функционально все то же самое, происходит считывание команды и вывод, если нужно, результата на экран. Поиск k-по возрастанию элемента происходит в словаре узлов дерева простым сравниванием числа k (переменной «key») и атрибута «order».

```
tree = binTree()
with open('input.txt') as f:
    with open('output.txt', 'w') as f1:
        while True:
            command = f.readline()
            if command == '':
                break
            command = list(map(str, command.split()))
            key = int(command[1])
            command = command[0]
            if command == '+':
                tree.insert(key)
            else:
                for i in tree.nodes:
                    if tree.nodes[i].order == key:
                        print(tree.nodes[i].key)
```

Задание 5.

Реализуйте простое двоичное дерево поиска.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание операций с деревом, их количество N не превышает 100. В каждой строке находится одна из следующих операций:

  – insert x – добавить в дерево ключ x. Если ключ x есть в дереве, то ничего делать не надо;

  – delete x – удалить из дерева ключ x. Если ключа x в дереве нет, то ничего делать не надо;

  – exists x – если ключ x есть в дереве выведите «true», если нет – «false»;

  – next x – выведите минимальный элемент в дереве, строго больший x, или «none», если такого нет;

  – prev x – выведите максимальный элемент в дереве, строго меньший x, или «none», если такого нет.

  В дерево помещаются и извлекаются только целые числа, не превышающие по модулю 10^9.
- **Ограничения на входные данные.** $0 \le N \le 100$, $|x_i| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите последовательно результат выполнения всех операций exists, next, prev. Следуйте формату выходного файла из примера.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 512 Мб.

Решение:

Конструкторы классов и функция «insert» такие же, как в задании №3.

```
class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

```
    def insert(self, key):
        if key not in self.nodes:
            self.nodes[key] = node()
            self.nodes[key].key = key
            if not tree.root:
                tree.root = self.nodes[key]
            else:
                root = tree.root
                while True:
                    if key < root.key:
                        if not root.left:
                            root.left = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.left
                    else:
                        if not root.right:
                            root.right = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.right
```

Далее добавляется функция «next». Она является копией функции «findMinMore» из задания №3.

```
    def next(self, key):
        min = float('inf')
        for j in self.nodes:
            if self.nodes[j].key > key and key < min:
                min = self.nodes[j].key
        if min == float('inf'):
            print('none')
        else:
            print(min)
```

Также появляется функция «prev» - полная противоположность предыдущей функции.

```
    def prev(self, key):
        max = float('-inf')
        for j in self.nodes:
            if self.nodes[j].key < key and key > max:
                max = self.nodes[j].key
        if max == float('-inf'):
            print('none')
        else:
            print(max)
```

Далее идет функция «nextNode», отвечающая за нахождение узла, который встанет на место удаляемого.

```
60      def nextNode(self, node):
61          if not node.right:
62              return node.right
63          if not node.right.left:
64              return node.right
65          node = node.right
66          while node.left:
67              node = node.left
68          return node
```

Далее на скриншотах изображена функция удаления узла. В общем, случая всего три:

1) Узла на замену из правого поддерева не нашлось;
2) Заменой является правый ребенок удаляемого узла;
3) Заменой является самый левый потомок правого ребенка узла.

Первый случай:

```
70      def delete(self, key):
71          if key in self.nodes:
72              node = self.nodes[key]
73              replace = self.nextNode(node)
74              if not replace:
75                  if not node.parent:
76                      if not node.left:
77                          self.root = None
78                      else:
79                          node.left.parent = None
80                          self.root = node.left
81                  else:
82                      if not node.left:
83                          if node.parent.left == node:
84                              node.parent.left = None
85                          else:
86                              node.parent.right = None
87                      else:
88                          if node.parent.left == node:
89                              node.parent.left = node.left
90                          else:
91                              node.parent.right = node.left
92                          node.left.parent = node.parent
```

Второй случай:

```
93              elif replace == node.right:
94                  if not node.parent:
95                      replace.parent = None
96                      if node.left:
97                          replace.left = node.left
98                          node.left.parent = replace
99                      self.root = replace
100                 else:
101                     if node.parent.left == node:
102                         node.parent.left = replace
103                     else:
104                         node.parent.right = replace
105                     replace.parent = node.parent
106                     if node.left:
107                         replace.left = node.left
108                         node.left.parent = replace
```

И третий случай (на двух скриншотах):

```
109             else:
110                 if not node.parent:
111                     if not node.left:
112                         if not replace.right:
113                             replace.parent.left = None
114                         else:
115                             replace.parent.left = replace.right
116                             replace.right.parent = replace.parent
117                         replace.right = node.right
118                         node.right.parent = replace
119                         replace.parent = None
120                         self.root = replace
121                     else:
122                         if not replace.right:
123                             replace.parent.left = None
124                         else:
125                             replace.parent.left = replace.right
126                             replace.right.parent = replace.parent
127                         replace.right = node.right
128                         node.right.parent = replace
129                         replace.left = node.left
130                         node.left.parent = replace
131                         replace.parent = None
132                         self.root = replace
```

```
                else:
                    if not node.left:
                        if not replace.right:
                            replace.parent.left = None
                        else:
                            replace.parent.left = replace.right
                            replace.right.parent = replace.parent
                        replace.right = node.right
                        node.right.parent = replace
                        replace.parent = node.parent
                        if node.parent.left == node:
                            node.parent.left = replace
                        else:
                            node.parent.right = replace
                    else:
                        if not replace.right:
                            replace.parent.left = None
                        else:
                            replace.parent.left = replace.right
                            replace.right.parent = replace.parent
                        replace.right = node.right
                        node.right.parent = replace
                        replace.left = node.left
                        node.left.parent = replace
                        replace.parent = node.parent
            self.nodes.pop(key)
```

В основном коде происходит считывание очередной команды и вывод, если нужно, ее результата, на экран. Наличие ключа в дерева проверяется наличием в словаре этого ключа.

```
tree = binTree()
with open('input.txt') as f:
    while True:
        command = f.readline()
        if command == '':
            break
        command = list(map(str, command.split()))
        key = int(command[1])
        command = command[0]
        if command == 'insert':
            tree.insert(key)
        elif command == 'delete':
            tree.delete(key)
        elif command == 'exists':
            if key in tree.nodes:
                print('true')
            else:
                print('false')
        elif command == 'next':
            tree.next(key)
        elif command == 'prev':
            tree.prev(key)
```

I. Ввод всего дерева сразу и дальнейшая работа с ним

Задание 1.

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 != −1 и R_i != −1, то L_i != 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.

Решение:

Конструктор классов идентичен предыдущим заданиям, только добавлен атрибут «passed», нужный для обхода дерева (понадобится для определения, был текущий узел пройден или нет).

```
class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.passed = False


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

Первая функция – прямой обход дерева. Принцип таков: если есть левый непройденный ребенок, то путь идет в левое поддерево. Иначе, если есть правый непройденный ребенок, то выводится ключ текущего узла и он обозначается пройденным. Путь идет в правое поддерево. Если не выполняется ни то, ни другое, происходит проверка текущего узла на пройденность, а затем подъем до первого непройденного узла или корня дерева.

```
    def in_order(self):
        node = self.root
        passed = not node.passed
        while True:
            if node.left and node.left.passed != passed:
                node = node.left
            elif node.right and node.right.passed != passed:
                print(node.key, end=' ')
                node.passed = passed
                node = node.right
            else:
                if node.passed != passed:
                    print(node.key, end=' ')
                    node.passed = passed
                if node != self.root:
                    while node.passed == passed:
                        node = node.parent
                        if node == self.root:
                            break
                else:
                    break
```

Далее идет функция «pre_order», поперечный обход дерева. Если есть непройденный левый ребенок, то печатается ключ текущего узла и путь идет в левое поддерево. Если левый ребенок пройден, а правый нет, то путь идет в правое поддерево. Иначе, текущий узел обозначается пройденным и происходит подъем на один узел наверх, до корня. Если же левого ребенка нет, а правый не пройден, то путь идет в правое поддерево. Если левого ребенка нет, а правый ребенок пройде или же

его не существует, то печатается ключ текущего узла и происходит подъем на один узел наверх, до корня.

```
    def pre_order(self):
        node = self.root
        passed = not node.passed
        while True:
            if node.left:
                if node.left.passed != passed:
                    print(node.key, end=' ')
                    node = node.left
                elif node.right and node.right.passed != passed:
                    node = node.right
                else:
                    node.passed = passed
                    if node == self.root:
                        break
                    else:
                        node = node.parent
            else:
                if node.right:
                    if node.right.passed != passed:
                        print(node.key, end=' ')
                        node = node.right
                    else:
                        node.passed = passed
                        if node == self.root:
                            break
                        else:
                            node = node.parent
                else:
                    print(node.key, end=' ')
                    node.passed = passed
                    if node == self.root:
                        break
                    else:
                        node = node.parent
```

Следующий на очереди – обратный обход. Здесь если есть левый непройденный ребенок, то путь идет в левое поддерево. Иначе, если есть правый непройденный ребенок, то путь идет в правое поддерево. Только после этого печатается ключ текущего узла и происходит подъем на один узел наверх, вплоть до корня.

```
    def post_order(self):
        node = self.root
        passed = not node.passed
        while True:
            if node.left:
                if node.left.passed != passed:
                    node = node.left
                elif node.right and node.right.passed != passed:
                    node = node.right
                else:
                    print(node.key, end=' ')
                    node.passed = passed
                    if node == self.root:
                        break
                    else:
                        node = node.parent
            else:
                if node.right:
                    if node.right.passed != passed:
                        node = node.right
                    else:
                        print(node.key, end=' ')
                        node.passed = passed
                        if node == self.root:
                            break
                        else:
                            node = node.parent
                else:
                    print(node.key, end=' ')
                    node.passed = passed
                    if node == self.root:
                        break
                    else:
                        node = node.parent
```

В основном коде создание дерева проходит следующим образом: сначала создаются все узлы дерева с их ключами, а уже потом им присваиваются родители и дети. Таким образом исключается возможность потери какого-либо узла дерева.

```
with open('input.txt') as f:
    n = int(f.readline())
    tree = binTree()
    data = []
    for i in range(0, n):
        data.append(list(map(int, f.readline().split())))
        tree.nodes[i] = node()
        tree.nodes[i].key = data[i][0]

for i in range(0, n):
    if data[i][1] != -1:
        tree.nodes[i].left = tree.nodes[data[i][1]]
        tree.nodes[data[i][1]].parent = tree.nodes[i]
    if data[i][2] != -1:
        tree.nodes[i].right = tree.nodes[data[i][2]]
        tree.nodes[data[i][2]].parent = tree.nodes[i]
    if i == 0:
        tree.root = tree.nodes[i]
```

В конце же происходит вызов функций обхода дерева и вывод их на экран.

```
130  tree.in_order()
131  print()
132  tree.pre_order()
133  print()
134  tree.post_order()
```

Задание 6.

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 != −1 и R_i != −1, то L_i != 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.

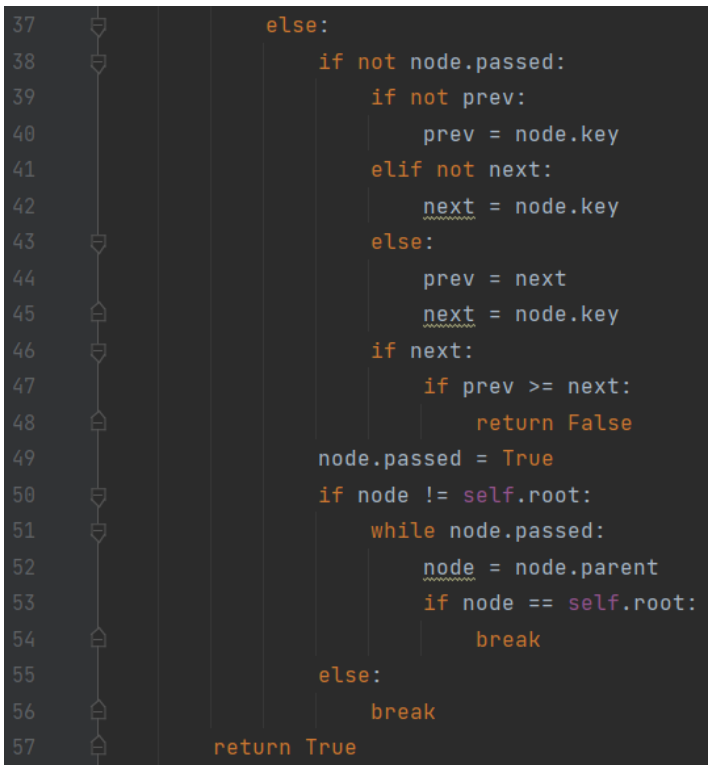
Решение:

Конструкторы классов такие же, есть атрибут «passed», поскольку в этой задаче понадобится прямой обход дерева.

```
class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.passed = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

Нам понадобится единственная функция, опознающая двоичное дерево поиска. Здесь происходит прямой обходит дерева с хранением двух ключей – предыдущего и следующего. Если в какой-то момент предыдущий ключ станет больше или равен следующему, это данное дерево не является деревом поиска.

```
    def is_binTree(self):
        node = self.root
        prev = None
        next = None
        while True:
            if node.left and not node.left.passed:
                node = node.left
            elif node.right and not node.right.passed:
                if not prev:
                    prev = node.key
                elif not next:
                    next = node.key
                else:
                    prev = next
                    next = node.key
                if next:
                    if prev >= next:
                        return False
                node.passed = True
                node = node.right
```



```
            else:
                if not node.passed:
                    if not prev:
                        prev = node.key
                    elif not next:
                        next = node.key
                    else:
                        prev = next
                        next = node.key
                    if next:
                        if prev >= next:
                            return False
                    node.passed = True
                    if node != self.root:
                        while node.passed:
                            node = node.parent
                            if node == self.root:
                                break
                    else:
                        break
        return True
```

Также при процессе создания дерева происходит частичное опознание двоичного дерева поиска. При создании связи родитель – ребенок ключ левого ребенка должен быть меньше ключа родителя, а ключ правого ребенка – больше ключа родителя. Иначе программа завершится и выдаст отрицательный результат. Также, если узлов в дереве не больше одного, то оно в любом случае будет двоичным деревом поиска.

```
with open('input.txt') as f:
    n = int(f.readline())
    if n <= 1:
        with open('output.txt', 'w') as f:
            f.write('CORRECT')
        exit()
    tree = binTree()
    data = []
    for i in range(0, n):
        data.append(list(map(int, f.readline().split())))
        tree.nodes[i] = node()
        tree.nodes[i].key = data[i][0]


for i in range(0, n):
    if data[i][1] != -1:
        tree.nodes[i].left = tree.nodes[data[i][1]]
        tree.nodes[data[i][1]].parent = tree.nodes[i]
        if tree.nodes[i].left.key >= tree.nodes[i].key:
            with open('output.txt', 'w') as f:
                f.write('INCORRECT')
                exit()
    if data[i][2] != -1:
        tree.nodes[i].right = tree.nodes[data[i][2]]
        tree.nodes[data[i][2]].parent = tree.nodes[i]
        if tree.nodes[i].key >= tree.nodes[i].right.key:
            with open('output.txt', 'w') as f:
                f.write('INCORRECT')
                exit()
    if i == 0:
        tree.root = tree.nodes[i]
```

В конце происходит полное опознание двоичного дерева и вывод результата на экран.

```
res = tree.is_binTree()
if res == True:
    print('CORRECT')
else:
    print('INCORRECT')
```

Задание 7.

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 != −1 и R_i != −1, то L_i != 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.

Решение:

Конструкторы класса абсолютны идентичны предыдущему заданию, но в нем нет атрибута «pass», поэтому сразу перейдем к функции опознания двоичного дерева. Но теперь для проверки мы будем идти от листьев дерева наверх. По пути мы будем класть в переменную «max», максимальный ключ поддерева, из которого идет подъем. На каждом встречающемся узле идет сравнение ключа этого узла и максимума поддерева, из которого идет подъем. Если ключ узла меньше и подъем происходит из правого поддерева, то это не BST. Также если ключ узла меньше или равен максимуму и подъем происходит из левого поддерева, то это тоже не BST

```
    def is_binTree(self, node):
        max_key = node.key
        while node.parent:
            parent = node.parent
            if parent.left == node:
                if parent.key <= max_key:
                    return False
            else:
                if parent.key > max_key:
                    return False
            max_key = max(max_key, parent.key)
            node = parent
        return True
```

В основном коде происходит считывание дерева, сбор листьев и проверка дерева.

```
with open('input.txt') as f:
    n = int(f.readline())
    if n <= 1:
        with open('output.txt', 'w') as f:
            f.write('CORRECT')
        exit()
    tree = binTree()
    data = []
    leaves = []
    for i in range(0, n):
        data.append(list(map(int, f.readline().split())))
        tree.nodes[i] = node()
        tree.nodes[i].key = data[i][0]
        if data[i-1][1] == -1 and data[i-1][2] == -1:
            leaves.append(i)
for i in range(0, n):
    if data[i][1] != -1:
        tree.nodes[i].left = tree.nodes[data[i][1]]
        tree.nodes[data[i][1]].parent = tree.nodes[i]
    if data[i][2] != -1:
        tree.nodes[i].right = tree.nodes[data[i][2]]
        tree.nodes[data[i][2]].parent = tree.nodes[i]
    if i == 0:
        tree.root = tree.nodes[i]

for i in leaves:
    res = tree.is_binTree(tree.nodes[i])
    if not res:
        print('INCORRECT')
        exit()
print('CORRECT')
```

Задание 8.

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.

Решение:

Конструкторы классов такие же, с добавлением атрибута «height», отвечающего за высоту узла.

```
class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.height = 0


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

Единственная нужная функция - увеличения высоты. На вход будут подаваться листья дерева.

```
    def set_height(self, node):
        node.height = 1
        while node.parent:
            parent = node.parent
            if parent.height <= node.height:
                parent.height = node.height + 1
            node = parent
```

Ввод тот же самый, что и в предыдущих заданиях этого блока. Только в отдельный массив заносятся листья дерева.

```
with open('input.txt') as f:
    n = int(f.readline())
    if n == 0:
        with open('output.txt', 'w') as f:
            f.write('0')
            exit()
    tree = binTree()
    data = []
    leaves = []
    for i in range(1, n+1):
        data.append(list(map(int, f.readline().split())))
        tree.nodes[i] = node()
        tree.nodes[i].key = data[i-1][0]
        if data[i-1][1] == 0 and data[i-1][2] == 0:
            leaves.append(i)


for i in range(1, n+1):
    if data[i-1][1] != 0:
        tree.nodes[i].left = tree.nodes[data[i-1][1]]
        tree.nodes[data[i-1][1]].parent = tree.nodes[i]
    if data[i-1][2] != 0:
        tree.nodes[i].right = tree.nodes[data[i-1][2]]
        tree.nodes[data[i-1][2]].parent = tree.nodes[i]
    if i == 1:
        tree.root = tree.nodes[i]
```

В конце происходит присвоение высоты всем узлам и вывод высоты корня дерева.

```
for i in leaves:
    tree.set_height(tree.nodes[i])
with open('output.txt', 'w') as f:
    f.write(str(tree.root.height))
```

Скрин прохождения тестов:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.187	115990528	3989144	6

Задание 9.

Дано некоторое двоичное дерево поиска. Также даны запросы на удаление из него вершин, имеющих заданные ключи, причем вершины удаляются целиком вместе со своими поддеревьями. После каждого запроса на удаление выведите число оставшихся вершин в дереве. В вершинах данного дерева записаны ключи – целые числа, по модулю не превышающие 10^9. Гарантируется, что данное дерево является

двоичным деревом поиска, в частности, 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.

Высота дерева не превосходит 25, таким образом, можно считать, что оно сбалансировано.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева и описание запросов на удаление. 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). Все ключи различны. Гарантируется, что данное дерево является деревом поиска. В следующей строке находится число M – число запросов на удаление. В следующей строке находятся M чисел, разделенных пробелами – ключи, вершины с которыми (вместе с их поддеревьями) необходимо удалить. Все эти числа не превосходят 10^9 по абсолютному значению. Вершина с таким ключом не обязана существовать в дереве – в этом случае дерево изменять не требуется. Гарантируется, что корень дерева никогда не будет удален.
- **Ограничения на входные данные.** $1 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$, $1 \le M \le 2 \cdot 10^5$.
- **Формат вывода / выходного файла (output.txt).** Выведите M строк. На i-ой строке требуется вывести число вершин, оставшихся в дереве после выполнения i-го запроса на удаление.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 256 Мб.

Решение:

Для этого задания нам понадобится очередь – реализация обхода в ширину. Конструкторы те же, добавлен атрибут «iter», отвечающий за изначальный индекс узла в файле.

```
from collections import deque

class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.iter = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

Единственная функция в классе – удаления поддерева. Сначала удаляется связь поддерева с родительским узлом. Далее происходит обход удаляемого поддерева в ширину. В очередь вносится корень удаляемого поддерева. Далее, пока очередь не окажется пустой, из нее удаляется первый внесенный из имеющихся узлов. Если у этого узла есть левый/правый ребенок, то он вносится в очередь. Сам узел удаляется из словаря узлов дерева. Далее в файл выводится количество узлов, оставшихся в дереве (длина словаря узлов).

```
    def delete_node_with_all(self, key):
        if key in self.nodes:
            root = self.nodes[key]
            if root.parent.left == root:
                root.parent.left = None
            else:
                root.parent.right = None
            delete_list = deque()
            delete_list.append(root)
            while len(delete_list) != 0:
                node = delete_list.popleft()
                if node.left:
                    delete_list.append(node.left)
                if node.right:
                    delete_list.append(node.right)
                self.nodes.pop(node.iter)
            global f
            f.write(str(len(self.nodes)) + '\n')
```

Ввод тот же самый, только добавляется ввод ключей удаляемых узлов и присвоение дополнительного атрибута «iter».

```
with open('input.txt') as f:
    n = int(f.readline())
    tree = binTree()
    data = []
    nodes_iter = {}
    for i in range(1, n+1):
        data.append(list(map(int, f.readline().split())))
        tree.nodes[i] = node()
        tree.nodes[i].key = data[i-1][0]
        tree.nodes[i].iter = i
        nodes_iter[data[i-1][0]] = i
    m = int(f.readline())
    for_delete = list(map(int, f.readline().split()))


for i in range(1, n+1):
    if data[i-1][1] != 0:
        tree.nodes[i].left = tree.nodes[data[i-1][1]]
        tree.nodes[data[i-1][1]].parent = tree.nodes[i]
    if data[i-1][2] != 0:
        tree.nodes[i].right = tree.nodes[data[i-1][2]]
        tree.nodes[data[i-1][2]].parent = tree.nodes[i]
    if i == 1:
        tree.root = tree.nodes[i]
```

В конце происходит преобразование ключа в индекс узла, удаление поддеревьев, если удаляемый узел существовал и вывод данных в файл.

```
with open('output.txt', 'w') as f:
    for j in for_delete:
        if j in nodes_iter:
            j = nodes_iter[j]
        else:
            j = 0
        if j in tree.nodes:
            tree.delete_node_with_all(j)
        else:
            f.write(str(len(tree.nodes)) + '\n')
```

Скрин прохождения тестов:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.515	141021184	6029382	1077960
1	OK	0.031	9646080	58	12
2	OK	0.031	9621504	27	12
3	OK	0.031	9678848	34	15
4	OK	0.031	9584640	211	30
5	OK	0.031	9641984	246	30
6	OK	0.031	9662464	3437	457
7	OK	0.031	9703424	3363	483
8	OK	0.031	10067968	18842	4247
9	OK	0.046	10219520	25683	3739
10	OK	0.046	11120640	69351	14791
11	OK	0.046	11833344	88936	11629
12	OK	0.062	14970880	244892	40297
13	OK	0.078	15036416	255614	37596
14	OK	0.234	30437376	978616	141281
15	OK	0.234	30384128	992647	137802
16	OK	0.546	57884672	2488583	634135
17	OK	0.812	83771392	3489729	483105
18	OK	1.015	102100992	4639039	1077960
19	OK	1.453	140402688	6007604	931260
20	OK	1.515	141021184	6029382	916969

Задание 10.

Свойство двоичного дерева поиска можно сформулировать следующим образом: для каждой вершины дерева выполняется следующее условие:

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

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

- **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 ($i < R_i \leq N$ или $R_i = 0$, если правого ребенка нет).

- **Ограничения на входные данные.** $0 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$.

На 60% от при $0 \leq N \leq 2000$.

- **Формат вывода / выходного файла (output.txt).** Выведите «YES», если данное во входном файле дерево является двоичным деревом поиска, и «NO», если не является.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 256 Мб.

Решение:

Задание абсолютно идентично заданию №6, поэтому различия есть только в вводе и выводе данных (поправка на индексы и вывод «YES»/«NO» вместо «CORRECT»/«INCORRECT»).

```
with open('input.txt') as f:
    n = int(f.readline())
    if n <= 1:
        with open('output.txt', 'w') as f:
            f.write('YES')
        exit()
    tree = binTree()
    data = []
    for i in range(1, n+1):
        data.append(list(map(int, f.readline().split())))
        tree.nodes[i] = node()
        tree.nodes[i].key = data[i-1][0]

for i in range(1, n+1):
    if data[i-1][1] != 0:
        tree.nodes[i].left = tree.nodes[data[i-1][1]]
        tree.nodes[data[i-1][1]].parent = tree.nodes[i]
        if tree.nodes[i].left.key >= tree.nodes[i].key:
            with open('output.txt', 'w') as f:
                f.write('NO')
                exit()
    if data[i-1][2] != 0:
        tree.nodes[i].right = tree.nodes[data[i-1][2]]
        tree.nodes[data[i-1][2]].parent = tree.nodes[i]
        if tree.nodes[i].key >= tree.nodes[i].right.key:
            with open('output.txt', 'w') as f:
                f.write('NO')
                exit()
    if i == 1:
        tree.root = tree.nodes[i]
```

```
res = tree.is_binTree()
with open('output.txt', 'w') as f:
    if res:
        f.write('YES')
    else:
        f.write('NO')
```

Скрин прохождения тестов:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла	Группа тестов
Max		1.109	102604800	3988813	3	

Задание 12.

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:

$$-1 \le B(V) \le 1$$

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.

- **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). Все ключи различны. Гарантируется, что данное дерево является деревом поиска.
- **Ограничения на входные данные.** $0 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Для i-ой вершины в i-ой строке выведите одно число – баланс данной вершины.

- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 256 Мб.

Решение:

Конструкторы те же самые, с атрибутом высоты и функцией ее определения.

```
class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.height = 0
        self.balance = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}

    def set_height(self, node):
        node.height = 1
        while node.parent:
            parent = node.parent
            if parent.height <= node.height:
                parent.height = node.height + 1
            node = parent
```

Ввод аналогичный предыдущим заданиям. Производится определение листов дерева и установление высоты узлов.

```python
with open('input.txt') as f:
    n = int(f.readline())
    if n == 0:
        with open('output.txt', 'w') as f:
            f.write('0')
            exit()
    tree = binTree()
    data = []
    leaves = []
    for i in range(1, n+1):
        data.append(list(map(int, f.readline().split())))
        tree.nodes[i] = node()
        tree.nodes[i].key = data[i-1][0]
        if data[i-1][1] == 0 and data[i-1][2] == 0:
            leaves.append(i)


for i in range(1, n+1):
    if data[i-1][1] != 0:
        tree.nodes[i].left = tree.nodes[data[i-1][1]]
        tree.nodes[data[i-1][1]].parent = tree.nodes[i]
    if data[i-1][2] != 0:
        tree.nodes[i].right = tree.nodes[data[i-1][2]]
        tree.nodes[data[i-1][2]].parent = tree.nodes[i]
    if i == 1:
        tree.root = tree.nodes[i]

for i in leaves:
    tree.set_height(tree.nodes[i])
```

Далее для всех узлов, в том порядке, в котором они были введены, считается их баланс и выводится в файл.

```python
with open('output.txt', 'w') as f:
    for i in range(1, n+1):
        node = tree.nodes[i]
        if node.left:
            if node.right:
                f.write(str(node.right.height - node.left.height) + '\n')
            else:
                f.write(str(-node.left.height) + '\n')
        else:
            if node.right:
                f.write(str(node.right.height) + '\n')
            else:
                f.write('0' + '\n')
```

Скрин прохождения тестов:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.500	115867648	3986010	1688889

Задание 13.

Дано дерево, в котором баланс корня равен 2. Сделайте левый поворот.

- **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). Все ключи различны. Гарантируется, что данное дерево является деревом поиска. Баланс корня дерева (вершины с номером 1) равен 2, баланс всех остальных вершин находится в пределах от -1 до 1.
- **Ограничения на входные данные.** $3 \le N \le 2 \cdot 10^5$, $|K_i| \le 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления левого поворота. Нумерация вершин может быть произвольной при условии соблюдения формата. Так, номер вершины должен быть меньше номера ее детей.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 256 Мб.

Решение:

Конструкторы те же, добавлен атрибут «balance», также здесь используется очередь для обхода в ширину.

```python
from collections import deque

class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.height = 0
        self.balance = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}

    def set_height(self, node):
        node.height = 1
        while node.parent:
            parent = node.parent
            if parent.height <= node.height:
                parent.height = node.height + 1
            else:
                break
            node = parent
```

Следующая функция – установка баланса всех узлов (аналогично основному коду в задании №12).

```python
    def set_balance(self):
        for i in range(1, n + 1):
            node = self.nodes[i]
            if node.left:
                if node.right:
                    node.balance = node.right.height - node.left.height
                else:
                    node.balance = -node.left.height
            else:
                if node.right:
                    node.balance = node.right.height
                else:
                    node.balance = 0
```

Функция левого поворота – согласно схеме, указанной в задании. Два случая – баланс правого ребенка равен -1 и не равен -1.

```
    def left_turn(self, A):
        B = A.right
        if B.balance == -1:
            C = B.left
            X = C.left
            Y = C.right
            if X:
                X.parent = A
            A.right = X
            if Y:
                Y.parent = B
            B.left = Y
            if A.key != self.root.key:
                C.parent = A.parent
                if A.parent.left == A:
                    A.parent.left = C
                else:
                    A.parent.right = C
            else:
                self.root = C
            A.parent = C
            C.left = A
            B.parent = C
            C.right = B
        else:
            Y = B.left
            if Y:
                Y.parent = A
            A.right = Y
            if A != self.root:
                B.parent = A.parent
                if A.parent.left == A:
                    A.parent.left = B
                else:
                    A.parent.right = B
            else:
                self.root = B
            A.parent = B
            B.left = A
```

Следующая функция – написана для вывода получившегося после левого поворота дерева. Здесь используется обход в ширину с помощью очереди (рассмотрен в задании №9). Только здесь используется счетчик «i» для индексации узлов и их детей.

```
84      def tree_output(self):
85          q = deque()
86          q.append(self.root)
87          i = 1
88          with open('output.txt', 'w') as f:
89              global n
90              f.write(str(n) + '\n')
91              while len(q) != 0:
92                  node = q.popleft()
93                  f.write(str(node.key))
94                  if node.left:
95                      i += 1
96                      f.write(' ' + str(i))
97                      q.append(node.left)
98                  else:
99                      f.write(' 0')
100                 if node.right:
101                     i += 1
102                     f.write(' ' + str(i) + '\n')
103                     q.append(node.right)
104                 else:
105                     f.write(' 0\n')
```

Ввод данных и установка высоты узлов абсолютно те же. Поэтому перейдем к финальной части кода. Здесь для всех узлов высчитывается баланс. Далее происходит левый поворот и запись получившегося дерева в файл.

```
136 tree.set_balance()
137 tree.left_turn(tree.root)
138 tree.tree_output()
```

Скрин прохождения тестов:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.734	112599040	3986416	3986416

Задание 14.

- **Формат ввода / входного файла (input.txt).** Входной файл содержит описание двоичного дерева, а также ключа вершины, которую требуется вставить в дерево. 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 ($i < R_i \leq N$ или $R_i = 0$, если правого ребенка нет). Все ключи различны. Гарантируется, что данное дерево является корректным АВЛ-деревом. В последней строке содержится число X – ключ вершины, которую требуется вставить в дерево. Гарантируется, что такой вершины в дереве нет.

- **Ограничения на входные данные.** $0 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$, $|X| \leq 10^9$.
- **Формат вывода / выходного файла (output.txt).** Выведите в том же формате дерево после осуществления операции вставки. Нумерация вершин может быть произвольной при условии соблюдения формата.
- **Ограничение по времени.** 2 сек.
- **Ограничение по памяти.** 256 Мб.

Решение:

Конструкторы классов те же, добавлен атрибут «balance» для хранения баланса узла. Также импортирована очередь для обхода в ширину и вывода конечного дерева в файл.

```
from collections import deque

class node:

    def __init__(self):
        self.key = None
        self.left = None
        self.right = None
        self.parent = None
        self.height = 0
        self.balance = None


class binTree:

    def __init__(self):
        self.root = None
        self.nodes = {}
```

Для вставки элемента в АВЛ-дерево используется функция из задания №3.

```
    def insert(self, key):
        if key not in self.nodes:
            self.nodes[key] = node()
            self.nodes[key].key = key
            if not self.root:
                self.root = self.nodes[key]
            else:
                root = self.root
                while True:
                    if key < root.key:
                        if not root.left:
                            root.left = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.left
                    else:
                        if not root.right:
                            root.right = self.nodes[key]
                            self.nodes[key].parent = root
                            break
                        else:
                            root = root.right
```

Знакомые нам функции установления высоты узлов и значения их баланса.

```
    def set_height(self, node):
        node.height = 1
        while node.parent:
            parent = node.parent
            if parent.height <= node.height:
                parent.height = node.height + 1
            else:
                break
            node = parent

    def set_balance(self):
        for i in range(1, n + 1):
            node = self.nodes[i]
            if node.left:
                if node.right:
                    node.balance = node.right.height - node.left.height
                else:
                    node.balance = -node.left.height
            else:
                if node.right:
                    node.balance = node.right.height
                else:
                    node.balance = 0
```

Функция левого поворота абсолютно та же, что и в задании №13, так что приведем функцию правого поворота, которая противоположна левому повороту.

```
    def right_turn(self, A):
        B = A.left
        if B.balance == 1:
            C = B.right
            X = C.left
            Y = C.right
            if X:
                X.parent = B
            B.right = X
            if Y:
                Y.parent = A
            A.left = Y
            if A.key != self.root.key:
                C.parent = A.parent
                if A.parent.left == A:
                    A.parent.left = C
                else:
                    A.parent.right = C
            else:
                self.root = C
            A.parent = C
            C.right = A
            B.parent = C
            C.left = B
        else:
            Y = B.right
            if Y:
                Y.parent = A
            A.left = Y
            if A != self.root:
                B.parent = A.parent
                if A.parent.left == A:
                    A.parent.left = B
                else:
                    A.parent.right = B
            else:
                self.root = B
            A.parent = B
            B.right = A
```

Функция вывода получившегося дерева в файл абсолютно такая же, как и в задании №13, с использованием обхода в ширину с помощью очереди.

```
    def tree_output(self):
        q = deque()
        q.append(self.root)
        i = 1
        with open('output.txt', 'w') as f:
            global n
            f.write(str(n+1) + '\n')
            while len(q) != 0:
                node = q.popleft()
                f.write(str(node.key))
                if node.left:
                    i += 1
                    f.write(' ' + str(i))
                    q.append(node.left)
                else:
                    f.write(' 0')
                if node.right:
                    i += 1
                    f.write(' ' + str(i) + '\n')
                    q.append(node.right)
                else:
                    f.write(' 0\n')
```

Принципы ввода и определения высоты узлов такие же, только есть исключение при пустом дереве.

```
with open('input.txt') as f:
    n = int(f.readline())
    tree = binTree()
    if n != 0:
        data = []
        leaves = []
        for i in range(1, n+1):
            data.append(list(map(int, f.readline().split())))
            tree.nodes[i] = node()
            tree.nodes[i].key = data[i-1][0]
            if data[i-1][1] == 0 and data[i-1][2] == 0:
                leaves.append(i)
    insert_key = int(f.readline())

if n != 0:
    for i in range(1, n+1):
        if data[i-1][1] != 0:
            tree.nodes[i].left = tree.nodes[data[i-1][1]]
            tree.nodes[data[i-1][1]].parent = tree.nodes[i]
        if data[i-1][2] != 0:
            tree.nodes[i].right = tree.nodes[data[i-1][2]]
            tree.nodes[data[i-1][2]].parent = tree.nodes[i]
        if i == 1:
            tree.root = tree.nodes[i]

    for i in leaves:
        tree.set_height(tree.nodes[i])
```

В конце в дерево соответственно вставляется ключ. Далее идет перерасчет высоты узлов дерева, относительно вставленного узла. Затем

программа идет вверх от вставленного узла. Если встречается дисбаланс какой-либо вершины, то делается соответствующий поворот и подъем прекращается. Далее полученное дерево выводится в файл.

```
203     tree.insert(insert_key)
204     tree.set_height(tree.nodes[insert_key])
205     node = tree.nodes[insert_key].parent
206     if node:
207         while True:
208             if node.left:
209                 if node.right:
210                     node.balance = node.right.height - node.left.height
211                 else:
212                     node.balance = -node.left.height
213             else:
214                 if node.right:
215                     node.balance = node.right.height
216                 else:
217                     node.balance = 0
218             if node.balance == 2:
219                 tree.left_turn(node)
220                 break
221             elif node.balance == -2:
222                 tree.right_turn(node)
223                 break
224             if node == tree.root:
225                 break
226             node = node.parent
227     tree.tree_output()
```

Скрин прохождения тестов:

Верное решение!
Результаты работы Вашего решения

№ теста	Результат	Время, с	Память	Размер входного файла	Размер выходного файла
Max		1.765	112652288	4011957	4011966

Вывод:

Была изучена и применена на практике такая структура данных, как двоичное дерево поиска. Рассмотрены различные алгоритмы ввода и работы с деревьями поиска.