

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

Решение:

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

```
1 class node:
2
3     def __init__(self):
4         self.key = None
5         self.left = None
6         self.right = None
7         self.parent = None
8
9
10 class binTree:
11
12     def __init__(self):
13         self.root = None
14         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.

```
16 def insert(self, key):
17     if key not in self.nodes:
18         self.nodes[key] = node()
19         self.nodes[key].key = key
20         if not tree.root:
21             tree.root = self.nodes[key]
22         else:
23             root = tree.root
24             while True:
25                 if key < root.key:
26                     if not root.left:
27                         root.left = self.nodes[key]
28                         self.nodes[key].parent = root
29                         break
30                     else:
31                         root = root.left
32                 else:
33                     if not root.right:
34                         root.right = self.nodes[key]
35                         self.nodes[key].parent = root
36                         break
37                     else:
38                         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.

```
40 def findMinMore(self, key):
41     min = float('inf')
42     for j in self.nodes:
43         if self.nodes[j].key > key and key < min:
44             min = self.nodes[j].key
45     if min == float('inf'):
46         print(0)
47     else:
48         print(min)
```

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

```

50     tree = binTree()
51     with open('input.txt') as f:
52         while True:
53             command = f.readline()
54             if command == '':
55                 break
56             command = list(map(str, command.split()))
57             key = int(command[1])
58             command = command[0]
59             if command == '+':
60                 tree.insert(key)
61             else:
62                 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.** $1 \leq x \leq 10^9$, $1 \leq N \leq 300000$, 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.

```

1  class node:
2
3      def __init__(self):
4          self.key = None
5          self.left = None
6          self.right = None
7          self.parent = None
8          self.order = None
9
10
11 class binTree:
12
13     def __init__(self):
14         self.root = None
15         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.

```
17 def insert(self, key):
18     if key not in self.nodes:
19         self.nodes[key] = node()
20         self.nodes[key].key = key
21         self.nodes[key].order = 1
22         if not tree.root:
23             tree.root = self.nodes[key]
24         else:
25             root = tree.root
26             while True:
27                 if key < root.key:
28                     if not root.left:
29                         root.left = self.nodes[key]
30                         self.nodes[key].parent = root
31                         break
32                     else:
33                         root = root.left
34                 else:
35                     if not root.right:
36                         root.right = self.nodes[key]
37                         self.nodes[key].parent = root
38                         break
39                     else:
40                         root = root.right
41             for i in self.nodes:
42                 if self.nodes[i].key > self.nodes[key].key:
43                     self.nodes[i].order += 1
44             elif i != key:
45                 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 сравнением числа k (переменной «key») и атрибута «order».

```
49 tree = binTree()
50 with open('input.txt') as f:
51     with open('output.txt', 'w') as f1:
52         while True:
53             command = f.readline()
54             if command == '':
55                 break
56             command = list(map(str, command.split()))
57             key = int(command[1])
58             command = command[0]
59             if command == '+':
60                 tree.insert(key)
61             else:
62                 for i in tree.nodes:
63                     if tree.nodes[i].order == key:
64                         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.** $0 \leq N \leq 100$, $|x_i| \leq 10^9$.
- **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.

```
1 class node:
2
3     def __init__(self):
4         self.key = None
5         self.left = None
6         self.right = None
7         self.parent = None
8
9
10 class binTree:
11
12     def __init__(self):
13         self.root = None
14         self.nodes = {}
```

```

16 def insert(self, key):
17     if key not in self.nodes:
18         self.nodes[key] = node()
19         self.nodes[key].key = key
20         if not tree.root:
21             tree.root = self.nodes[key]
22         else:
23             root = tree.root
24             while True:
25                 if key < root.key:
26                     if not root.left:
27                         root.left = self.nodes[key]
28                         self.nodes[key].parent = root
29                         break
30                     else:
31                         root = root.left
32                 else:
33                     if not root.right:
34                         root.right = self.nodes[key]
35                         self.nodes[key].parent = root
36                         break
37                     else:
38                         root = root.right

```

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

```

40 def next(self, key):
41     min = float('inf')
42     for j in self.nodes:
43         if self.nodes[j].key > key and key < min:
44             min = self.nodes[j].key
45     if min == float('inf'):
46         print('none')
47     else:
48         print(min)

```

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

```

50 def prev(self, key):
51     max = float('-inf')
52     for j in self.nodes:
53         if self.nodes[j].key < key and key > max:
54             max = self.nodes[j].key
55     if max == float('-inf'):
56         print('none')
57     else:
58         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 поддеревя 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

```

```

133         else:
134             if not node.left:
135                 if not replace.right:
136                     replace.parent.left = None
137                 else:
138                     replace.parent.left = replace.right
139                     replace.right.parent = replace.parent
140             replace.right = node.right
141             node.right.parent = replace
142             replace.parent = node.parent
143             if node.parent.left == node:
144                 node.parent.left = replace
145             else:
146                 node.parent.right = replace
147         else:
148             if not replace.right:
149                 replace.parent.left = None
150             else:
151                 replace.parent.left = replace.right
152                 replace.right.parent = replace.parent
153             replace.right = node.right
154             node.right.parent = replace
155             replace.left = node.left
156             node.left.parent = replace
157             replace.parent = node.parent
158     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.

```

161     tree = binTree()
162     with open('input.txt') as f:
163         while True:
164             command = f.readline()
165             if command == '':
166                 break
167             command = list(map(str, command.split()))
168             key = int(command[1])
169             command = command[0]
170             if command == 'insert':
171                 tree.insert(key)
172             elif command == 'delete':
173                 tree.delete(key)
174             elif command == 'exists':
175                 if key in tree.nodes:
176                     print('true')
177                 else:
178                     print('false')
179             elif command == 'next':
180                 tree.next(key)
181             elif command == 'prev':
182                 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.** $1 \leq n \leq 10^5$, $0 \leq K_i \leq 10^9$, $-1 \leq L_i, R_i \leq n-1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \neq -1$ и $R_i \neq -1$, то $L_i \neq R_i$. 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).

```

1 class node:
2
3     def __init__(self):
4         self.key = None
5         self.left = None
6         self.right = None
7         self.parent = None
8         self.passed = False
9
10
11 class binTree:
12
13     def __init__(self):
14         self.root = None
15         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.

```

17     def in_order(self):
18         node = self.root
19         passed = not node.passed
20         while True:
21             if node.left and node.left.passed != passed:
22                 node = node.left
23             elif node.right and node.right.passed != passed:
24                 print(node.key, end=' ')
25                 node.passed = passed
26                 node = node.right
27             else:
28                 if node.passed != passed:
29                     print(node.key, end=' ')
30                     node.passed = passed
31                 if node != self.root:
32                     while node.passed == passed:
33                         node = node.parent
34                         if node == self.root:
35                             break
36                 else:
37                     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 вверх, 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 вверх, до корня.

```
39 def pre_order(self):
40     node = self.root
41     passed = not node.passed
42     while True:
43         if node.left:
44             if node.left.passed != passed:
45                 print(node.key, end=' ')
46                 node = node.left
47             elif node.right and node.right.passed != passed:
48                 node = node.right
49             else:
50                 node.passed = passed
51                 if node == self.root:
52                     break
53                 else:
54                     node = node.parent
55         else:
56             if node.right:
57                 if node.right.passed != passed:
58                     print(node.key, end=' ')
59                     node = node.right
60                 else:
61                     node.passed = passed
62                     if node == self.root:
63                         break
64                     else:
65                         node = node.parent
66             else:
67                 print(node.key, end=' ')
68                 node.passed = passed
69                 if node == self.root:
70                     break
71                 else:
72                     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 вверх, вплоть до корня.

```

74 def post_order(self):
75     node = self.root
76     passed = not node.passed
77     while True:
78         if node.left:
79             if node.left.passed != passed:
80                 node = node.left
81             elif node.right and node.right.passed != passed:
82                 node = node.right
83             else:
84                 print(node.key, end=' ')
85                 node.passed = passed
86                 if node == self.root:
87                     break
88                 else:
89                     node = node.parent
90         else:
91             if node.right:
92                 if node.right.passed != passed:
93                     node = node.right
94             else:
95                 print(node.key, end=' ')
96                 node.passed = passed
97                 if node == self.root:
98                     break
99                 else:
100                     node = node.parent
101         else:
102             print(node.key, end=' ')
103             node.passed = passed
104             if node == self.root:
105                 break
106             else:
107                 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.

```

111 with open('input.txt') as f:
112     n = int(f.readline())
113     tree = binTree()
114     data = []
115     for i in range(0, n):
116         data.append(list(map(int, f.readline().split())))
117         tree.nodes[i] = node()
118         tree.nodes[i].key = data[i][0]
119
120     for i in range(0, n):
121         if data[i][1] != -1:
122             tree.nodes[i].left = tree.nodes[data[i][1]]
123             tree.nodes[data[i][1]].parent = tree.nodes[i]
124         if data[i][2] != -1:
125             tree.nodes[i].right = tree.nodes[data[i][2]]
126             tree.nodes[data[i][2]].parent = tree.nodes[i]
127         if i == 0:
128             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.** $0 \leq n \leq 10^5$, $-2^{31} \leq K_i \leq 2^{31} - 1$, $-1 \leq L_i, R_i \leq n - 1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \neq -1$ и $R_i \neq -1$, то $L_i \neq R_i$. 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.

```
1 class node:
2
3     def __init__(self):
4         self.key = None
5         self.left = None
6         self.right = None
7         self.parent = None
8         self.passed = None
9
10
11 class binTree:
12
13     def __init__(self):
14         self.root = None
15         self.nodes = {}
```

Нам понадобится единственная функция, опознающая двоичное дерево поиска. Здесь происходит прямой обход 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.

```
17 def is_binTree(self):
18     node = self.root
19     prev = None
20     next = None
21     while True:
22         if node.left and not node.left.passed:
23             node = node.left
24         elif node.right and not node.right.passed:
25             if not prev:
26                 prev = node.key
27             elif not next:
28                 next = node.key
29             else:
30                 prev = next
31                 next = node.key
32             if next:
33                 if prev >= next:
34                     return False
35             node.passed = True
36             node = node.right
```



```

37     else:
38         if not node.passed:
39             if not prev:
40                 prev = node.key
41             elif not next:
42                 next = node.key
43             else:
44                 prev = next
45                 next = node.key
46             if next:
47                 if prev >= next:
48                     return False
49             node.passed = True
50             if node != self.root:
51                 while node.passed:
52                     node = node.parent
53                     if node == self.root:
54                         break
55             else:
56                 break
57     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.

```

60 with open('input.txt') as f:
61     n = int(f.readline())
62     if n <= 1:
63         with open('output.txt', 'w') as f:
64             f.write('CORRECT')
65         exit()
66     tree = binTree()
67     data = []
68     for i in range(0, n):
69         data.append(list(map(int, f.readline().split())))
70         tree.nodes[i] = node()
71         tree.nodes[i].key = data[i][0]
72
73
74 for i in range(0, n):
75     if data[i][1] != -1:
76         tree.nodes[i].left = tree.nodes[data[i][1]]
77         tree.nodes[data[i][1]].parent = tree.nodes[i]
78         if tree.nodes[i].left.key >= tree.nodes[i].key:
79             with open('output.txt', 'w') as f:
80                 f.write('INCORRECT')
81             exit()
82     if data[i][2] != -1:
83         tree.nodes[i].right = tree.nodes[data[i][2]]
84         tree.nodes[data[i][2]].parent = tree.nodes[i]
85         if tree.nodes[i].key >= tree.nodes[i].right.key:
86             with open('output.txt', 'w') as f:
87                 f.write('INCORRECT')
88             exit()
89     if i == 0:
90         tree.root = tree.nodes[i]

```

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

```

92 res = tree.is_binTree()
93 if res == True:
94     print('CORRECT')
95 else:
96     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 поддеревья меньше ключа вершины V ;
- все ключи вершин из правого поддеревья 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.** $0 \leq n \leq 10^5$, $-2^{31} \leq K_i \leq 2^{31} - 1$, $-1 \leq L_i, R_i \leq n - 1$. Гарантируется, что данное дерево является двоичным деревом. В частности, если $L_i \neq -1$ и $R_i \neq -1$, то $L_i \neq R_i$. 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, поэтому сразу перейдем к функции опознания двоичного дерева. Используется тот же метод с использованием прямого обхода дерева. Только в теперь динамически хранятся предыдущий и следующий узлы, а не их ключи. Дерево считается не деревом поиска в двух случаях:

- 1) Если ключ предыдущего узла больше или равен ключу следующего, и следующий узел не является правым ребенком предыдущего;
- 2) Если ключ предыдущего узла больше ключа следующего, и следующий узел является правым ребенком предыдущего.

```

17 def is_binTree(self):
18     node = self.root
19     prev = None
20     next = None
21     while True:
22         if node.left and not node.left.passed:
23             node = node.left
24         elif node.right and not node.right.passed:
25             if not prev:
26                 prev = node
27             elif not next:
28                 next = node
29             else:
30                 prev = next
31                 next = node
32             if next:
33                 if prev.key >= next.key and prev.right != next:
34                     return False
35                 elif prev.key > next.key and prev.right == next:
36                     return False
37             node.passed = True
38             node = node.right
39     else:
40         if not node.passed:
41             if not prev:
42                 prev = node
43             elif not next:
44                 next = node
45             else:
46                 prev = next
47                 next = node
48             if next:
49                 if prev.key >= next.key and prev.right != next:
50                     return False
51                 elif prev.key > next.key and prev.right == next:
52                     return False
53             node.passed = True
54             if node != self.root:
55                 while node.passed:
56                     node = node.parent
57                     if node == self.root:
58                         break
59             else:
60                 break
61     return True

```

В основном коде происходит почти то же самое, что и в задании №6, с поправкой на разницу условий.

```

64 with open('input.txt') as f:
65     n = int(f.readline())
66     if n <= 1:
67         with open('output.txt', 'w') as f:
68             f.write('CORRECT')
69             exit()
70     tree = binTree()
71     data = []
72     for i in range(0, n):
73         data.append(list(map(int, f.readline().split())))
74         tree.nodes[i] = node()
75         tree.nodes[i].key = data[i][0]
76
77     for i in range(0, n):
78         if data[i][1] != -1:
79             tree.nodes[i].left = tree.nodes[data[i][1]]
80             tree.nodes[data[i][1]].parent = tree.nodes[i]
81             if tree.nodes[i].left.key >= tree.nodes[i].key:
82                 with open('output.txt', 'w') as f:
83                     f.write('INCORRECT')
84                     exit()
85         if data[i][2] != -1:
86             tree.nodes[i].right = tree.nodes[data[i][2]]
87             tree.nodes[data[i][2]].parent = tree.nodes[i]
88             if tree.nodes[i].key > tree.nodes[i].right.key:
89                 with open('output.txt', 'w') as f:
90                     f.write('INCORRECT')
91                     exit()
92         if i == 0:
93             tree.root = tree.nodes[i]
94
95     res = tree.is_binTree()
96     if res == True:
97         print('CORRECT')
98     else:
99         print('INCORRECT')

```

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 ($1 \leq i \leq N$)

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 ($i < L_i \leq N$ 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.** $0 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$. 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.

```
1 class node:
2
3     def __init__(self):
4         self.key = None
5         self.left = None
6         self.right = None
7         self.parent = None
8         self.height = 0
9
10
11 class binTree:
12
13     def __init__(self):
14         self.root = None
15         self.nodes = {}
```

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

```
17 def set_height(self, node):
18     node.height = 1
19     while node.parent:
20         parent = node.parent
21         if parent.height <= node.height:
22             parent.height = node.height + 1
23         node = parent
```

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

```

26 with open('input.txt') as f:
27     n = int(f.readline())
28     if n == 0:
29         with open('output.txt', 'w') as f:
30             f.write('0')
31             exit()
32     tree = binTree()
33     data = []
34     leaves = []
35     for i in range(1, n+1):
36         data.append(list(map(int, f.readline().split())))
37         tree.nodes[i] = node()
38         tree.nodes[i].key = data[i-1][0]
39         if data[i-1][1] == 0 and data[i-1][2] == 0:
40             leaves.append(i)
41
42
43     for i in range(1, n+1):
44         if data[i-1][1] != 0:
45             tree.nodes[i].left = tree.nodes[data[i-1][1]]
46             tree.nodes[data[i-1][1]].parent = tree.nodes[i]
47         if data[i-1][2] != 0:
48             tree.nodes[i].right = tree.nodes[data[i-1][2]]
49             tree.nodes[data[i-1][2]].parent = tree.nodes[i]
50         if i == 1:
51             tree.root = tree.nodes[i]

```

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

```

53     for i in leaves:
54         tree.set_height(tree.nodes[i])
55     with open('output.txt', 'w') as f:
56         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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ 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 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.** $1 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$, $1 \leq M \leq 2 \cdot 10^5$.
- **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.


```

1  from collections import deque
2
3  class node:
4
5      def __init__(self):
6          self.key = None
7          self.left = None
8          self.right = None
9          self.parent = None
10         self.iter = None
11
12
13  class binTree:
14
15      def __init__(self):
16          self.root = None
17          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).

```

19  def delete_node_with_all(self, key):
20      if key in self.nodes:
21          root = self.nodes[key]
22          if root.parent.left == root:
23              root.parent.left = None
24          else:
25              root.parent.right = None
26          delete_list = deque()
27          delete_list.append(root)
28          while len(delete_list) != 0:
29              node = delete_list.popleft()
30              if node.left:
31                  delete_list.append(node.left)
32              if node.right:
33                  delete_list.append(node.right)
34              self.nodes.pop(node.iter)
35          global f
36          f.write(str(len(self.nodes)) + '\n')

```

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

```

38 with open('input.txt') as f:
39     n = int(f.readline())
40     tree = binTree()
41     data = []
42     nodes_iter = {}
43     for i in range(1, n+1):
44         data.append(list(map(int, f.readline().split())))
45         tree.nodes[i] = node()
46         tree.nodes[i].key = data[i-1][0]
47         tree.nodes[i].iter = i
48         nodes_iter[data[i-1][0]] = i
49     m = int(f.readline())
50     for_delete = list(map(int, f.readline().split()))
51
52
53 for i in range(1, n+1):
54     if data[i-1][1] != 0:
55         tree.nodes[i].left = tree.nodes[data[i-1][1]]
56         tree.nodes[data[i-1][1]].parent = tree.nodes[i]
57     if data[i-1][2] != 0:
58         tree.nodes[i].right = tree.nodes[data[i-1][2]]
59         tree.nodes[data[i-1][2]].parent = tree.nodes[i]
60     if i == 1:
61         tree.root = tree.nodes[i]

```

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

```

64 with open('output.txt', 'w') as f:
65     for j in for_delete:
66         if j in nodes_iter:
67             j = nodes_iter[j]
68         else:
69             j = 0
70         if j in tree.nodes:
71             tree.delete_node_with_all(j)
72         else:
73             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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ 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»).

```
60 with open('input.txt') as f:
61     n = int(f.readline())
62     if n <= 1:
63         with open('output.txt', 'w') as f:
64             f.write('YES')
65         exit()
66     tree = binTree()
67     data = []
68     for i in range(1, n+1):
69         data.append(list(map(int, f.readline().split())))
70         tree.nodes[i] = node()
71         tree.nodes[i].key = data[i-1][0]
72
73     for i in range(1, n+1):
74         if data[i-1][1] != 0:
75             tree.nodes[i].left = tree.nodes[data[i-1][1]]
76             tree.nodes[data[i-1][1]].parent = tree.nodes[i]
77             if tree.nodes[i].left.key >= tree.nodes[i].key:
78                 with open('output.txt', 'w') as f:
79                     f.write('NO')
80                 exit()
81         if data[i-1][2] != 0:
82             tree.nodes[i].right = tree.nodes[data[i-1][2]]
83             tree.nodes[data[i-1][2]].parent = tree.nodes[i]
84             if tree.nodes[i].key >= tree.nodes[i].right.key:
85                 with open('output.txt', 'w') as f:
86                     f.write('NO')
87                 exit()
88     if i == 1:
89         tree.root = tree.nodes[i]
```

```

91     res = tree.is_binTree()
92     with open('output.txt', 'w') as f:
93         if res:
94             f.write('YES')
95         else:
96             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:

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

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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ 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 деревом поиска.

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

- **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.

```
1 class node:
2
3     def __init__(self):
4         self.key = None
5         self.left = None
6         self.right = None
7         self.parent = None
8         self.height = 0
9         self.balance = None
10
11
12 class binTree:
13
14     def __init__(self):
15         self.root = None
16         self.nodes = {}
17
18     def set_height(self, node):
19         node.height = 1
20         while node.parent:
21             parent = node.parent
22             if parent.height <= node.height:
23                 parent.height = node.height + 1
24             node = parent
```

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

```

26 with open('input.txt') as f:
27     n = int(f.readline())
28     if n == 0:
29         with open('output.txt', 'w') as f:
30             f.write('0')
31             exit()
32     tree = binTree()
33     data = []
34     leaves = []
35     for i in range(1, n+1):
36         data.append(list(map(int, f.readline().split())))
37         tree.nodes[i] = node()
38         tree.nodes[i].key = data[i-1][0]
39         if data[i-1][1] == 0 and data[i-1][2] == 0:
40             leaves.append(i)
41
42
43     for i in range(1, n+1):
44         if data[i-1][1] != 0:
45             tree.nodes[i].left = tree.nodes[data[i-1][1]]
46             tree.nodes[data[i-1][1]].parent = tree.nodes[i]
47         if data[i-1][2] != 0:
48             tree.nodes[i].right = tree.nodes[data[i-1][2]]
49             tree.nodes[data[i-1][2]].parent = tree.nodes[i]
50         if i == 1:
51             tree.root = tree.nodes[i]
52
53     for i in leaves:
54         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.

```

56 with open('output.txt', 'w') as f:
57     for i in range(1, n+1):
58         node = tree.nodes[i]
59         if node.left:
60             if node.right:
61                 f.write(str(node.right.height - node.left.height) + '\n')
62             else:
63                 f.write(str(-node.left.height) + '\n')
64         else:
65             if node.right:
66                 f.write(str(node.right.height) + '\n')
67             else:
68                 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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ 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 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.** $3 \leq N \leq 2 \cdot 10^5$, $|K_i| \leq 10^9$.
- **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.


```

1      from collections import deque
2
3      class node:
4
5          def __init__(self):
6              self.key = None
7              self.left = None
8              self.right = None
9              self.parent = None
10             self.height = 0
11             self.balance = None
12
13
14     class binTree:
15
16         def __init__(self):
17             self.root = None
18             self.nodes = {}
19
20         def set_height(self, node):
21             node.height = 1
22             while node.parent:
23                 parent = node.parent
24                 if parent.height <= node.height:
25                     parent.height = node.height + 1
26                 else:
27                     break
28             node = parent

```

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

```

30     def set_balance(self):
31         for i in range(1, n + 1):
32             node = self.nodes[i]
33             if node.left:
34                 if node.right:
35                     node.balance = node.right.height - node.left.height
36                 else:
37                     node.balance = -node.left.height
38             else:
39                 if node.right:
40                     node.balance = node.right.height
41                 else:
42                     node.balance = 0

```

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

```

44 def left_turn(self, A):
45     B = A.right
46     if B.balance == -1:
47         C = B.left
48         X = C.left
49         Y = C.right
50         if X:
51             X.parent = A
52             A.right = X
53         if Y:
54             Y.parent = B
55             B.left = Y
56         if A.key != self.root.key:
57             C.parent = A.parent
58             if A.parent.left == A:
59                 A.parent.left = C
60             else:
61                 A.parent.right = C
62         else:
63             self.root = C
64             A.parent = C
65             C.left = A
66             B.parent = C
67             C.right = B
68     else:
69         Y = B.left
70         if Y:
71             Y.parent = A
72             A.right = Y
73         if A != self.root:
74             B.parent = A.parent
75             if A.parent.left == A:
76                 A.parent.left = B
77             else:
78                 A.parent.right = B
79         else:
80             self.root = B
81             A.parent = B
82             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 ($1 \leq i \leq N$) 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 ($i < L_i \leq N$ 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.

```
1  from collections import deque
2
3  class node:
4
5      def __init__(self):
6          self.key = None
7          self.left = None
8          self.right = None
9          self.parent = None
10         self.height = 0
11         self.balance = None
12
13
14  class binTree:
15
16      def __init__(self):
17          self.root = None
18          self.nodes = {}
```

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

```

20 def insert(self, key):
21     if key not in self.nodes:
22         self.nodes[key] = node()
23         self.nodes[key].key = key
24         if not self.root:
25             self.root = self.nodes[key]
26         else:
27             root = self.root
28             while True:
29                 if key < root.key:
30                     if not root.left:
31                         root.left = self.nodes[key]
32                         self.nodes[key].parent = root
33                         break
34                     else:
35                         root = root.left
36                 else:
37                     if not root.right:
38                         root.right = self.nodes[key]
39                         self.nodes[key].parent = root
40                         break
41                     else:
42                         root = root.right

```

Знакомые нам функции установления высоты узлов и значения их баланса.

```

44 def set_height(self, node):
45     node.height = 1
46     while node.parent:
47         parent = node.parent
48         if parent.height <= node.height:
49             parent.height = node.height + 1
50         else:
51             break
52         node = parent
53
54 def set_balance(self):
55     for i in range(1, n + 1):
56         node = self.nodes[i]
57         if node.left:
58             if node.right:
59                 node.balance = node.right.height - node.left.height
60             else:
61                 node.balance = -node.left.height
62         else:
63             if node.right:
64                 node.balance = node.right.height
65             else:
66                 node.balance = 0

```

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

```

108 def right_turn(self, A):
109     B = A.left
110     if B.balance == 1:
111         C = B.right
112         X = C.left
113         Y = C.right
114         if X:
115             X.parent = B
116             B.right = X
117         if Y:
118             Y.parent = A
119             A.left = Y
120         if A.key != self.root.key:
121             C.parent = A.parent
122             if A.parent.left == A:
123                 A.parent.left = C
124             else:
125                 A.parent.right = C
126         else:
127             self.root = C
128         A.parent = C
129         C.right = A
130         B.parent = C
131         C.left = B
132     else:
133         Y = B.right
134         if Y:
135             Y.parent = A
136             A.left = Y
137         if A != self.root:
138             B.parent = A.parent
139             if A.parent.left == A:
140                 A.parent.left = B
141             else:
142                 A.parent.right = B
143         else:
144             self.root = B
145         A.parent = B
146         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.

```

149 def tree_output(self):
150     q = deque()
151     q.append(self.root)
152     i = 1
153     with open('output.txt', 'w') as f:
154         global n
155         f.write(str(n+1) + '\n')
156         while len(q) != 0:
157             node = q.popleft()
158             f.write(str(node.key))
159             if node.left:
160                 i += 1
161                 f.write(' ' + str(i))
162                 q.append(node.left)
163             else:
164                 f.write(' 0')
165             if node.right:
166                 i += 1
167                 f.write(' ' + str(i) + '\n')
168                 q.append(node.right)
169             else:
170                 f.write(' 0\n')

```

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

```

175 with open('input.txt') as f:
176     n = int(f.readline())
177     tree = binTree()
178     if n != 0:
179         data = []
180         leaves = []
181         for i in range(1, n+1):
182             data.append(list(map(int, f.readline().split())))
183             tree.nodes[i] = node()
184             tree.nodes[i].key = data[i-1][0]
185             if data[i-1][1] == 0 and data[i-1][2] == 0:
186                 leaves.append(i)
187         insert_key = int(f.readline())
188
189     if n != 0:
190         for i in range(1, n+1):
191             if data[i-1][1] != 0:
192                 tree.nodes[i].left = tree.nodes[data[i-1][1]]
193                 tree.nodes[data[i-1][1]].parent = tree.nodes[i]
194             if data[i-1][2] != 0:
195                 tree.nodes[i].right = tree.nodes[data[i-1][2]]
196                 tree.nodes[data[i-1][2]].parent = tree.nodes[i]
197             if i == 1:
198                 tree.root = tree.nodes[i]
199
200         for i in leaves:
201             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.