МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: АВЛ-деревья

Студент гр. 0383 Бояркин Н.А.

Преподаватель Берленко Т.А.

Санкт-Петербург

2021

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Бояркин Н.А.

Группа 0383

Тема работы: AVL-деревья, вставка и поиск. Демонстрация работы.

Исходные данные:

AVL-деревья, вставка и поиск. Демонстрация работы.

Содержание пояснительной записки:

«Содержание», «Введение», «Поиск в AVL-деревьях», «Вставка в AVL-деревьях», «Тестирование», «Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:

Не менее 20 страниц.

Дата выдачи задания: 01.11.2021

Дата сдачи реферата: 10.12.2021

Дата защиты реферата: 10.12.2021

Студент		Бояркин Н.А.
Преподаватель		Берленко Т.А.

АННОТАЦИЯ

Разработан класс узла (*class Node*) и класс AVL-дерева (*class AVLTree*). Реализованы функции вставки узла в дерево, а также функция поиска. Для реализации функции вставки были описаны функции для малого левого вращения и для малого правого вращения. Также метод для вычисления высоты вершины. Метод для вычисления *balance factor*. А также демонстрация всего дерева с помощью модуля pydot.

SUMMARY

A node class (class Node) and an AVL-tree class (class AVLTree) have been developed. The functions of inserting a node into a tree, as well as a search function, have been implemented. To implement the insert function, functions for small left rotation and small right rotation have been described. Also a method for calculating the height of the vertex. Method for calculating balance factor. And also a demonstration of the whole tree using the pydot module.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Цель работы: Изучить механизм работы AVL-деревьев, разработать вставку и поиск вершин и продемонстрировать их работу.

Задание: АВЛ-деревья - вставка и поиск. Демонстрация

Методы решение:

1. Разработать класс узла (*class Node*) и класс AVL-дерева (*class AVLTree*)
2. Реализовать функции вставки узла в дерево, а также функция поиска.
3. Для реализации функции вставки описать функции для малого левого вращения и для малого правого вращения.
4. Продемонстрировать работу.

1. ПОИСК В AVL-ДЕРЕВЬЯХ

1.1. Класс AVL-дерева

Для разработки AVL-деревьев был созданы два класса: *class Node* и *class AVLTree*. Класс *Node* имеет поля: *self.value* для хранения значения в вершине, *self.left* для хранения левого узла родителя, *self.right* для хранения правого узла родителя и *self.h* для хранения информации о том, на какой высоте находится вершина (В дальнейшем это будет использоваться при вычислении баланса вершины). Класс *AVLTree* имеет только одно поле - *self.visited* для хранения информации о том, какие узлы были пройдены во время выполнения функции поиска (В дальнейшем это будет использоваться для демонстрации метода поиска вершины в дереве).

1.2. Функция поиска

Реализован метод поиска - *search(root, value)*, где в качестве аргументов передается корень дерева и значение которое необходимо найти. Сначала происходит проверка является ли тип у корня None. И если является, то сделать список *self.visited* пустым. Это было сделано для того, чтобы учесть тот случай, когда мы в качестве аргумента передали значение вершины, которой в дереве нет, а так как функция *search(root, value)* является рекурсивной, то это условие является одной из ее точек выхода. Также точкой выхода является условие, когда мы нашли переданное значение в дереве. Далее мы проверяем является ли значение меньше значения у корня дерева. И в случае, если является, то мы идём влево и так далее по рекурсии, при этом записываем в список вершины, которые мы прошли, чтобы продемонстрировать ход поиска в другом методе.
Демонстрация функции поиска.

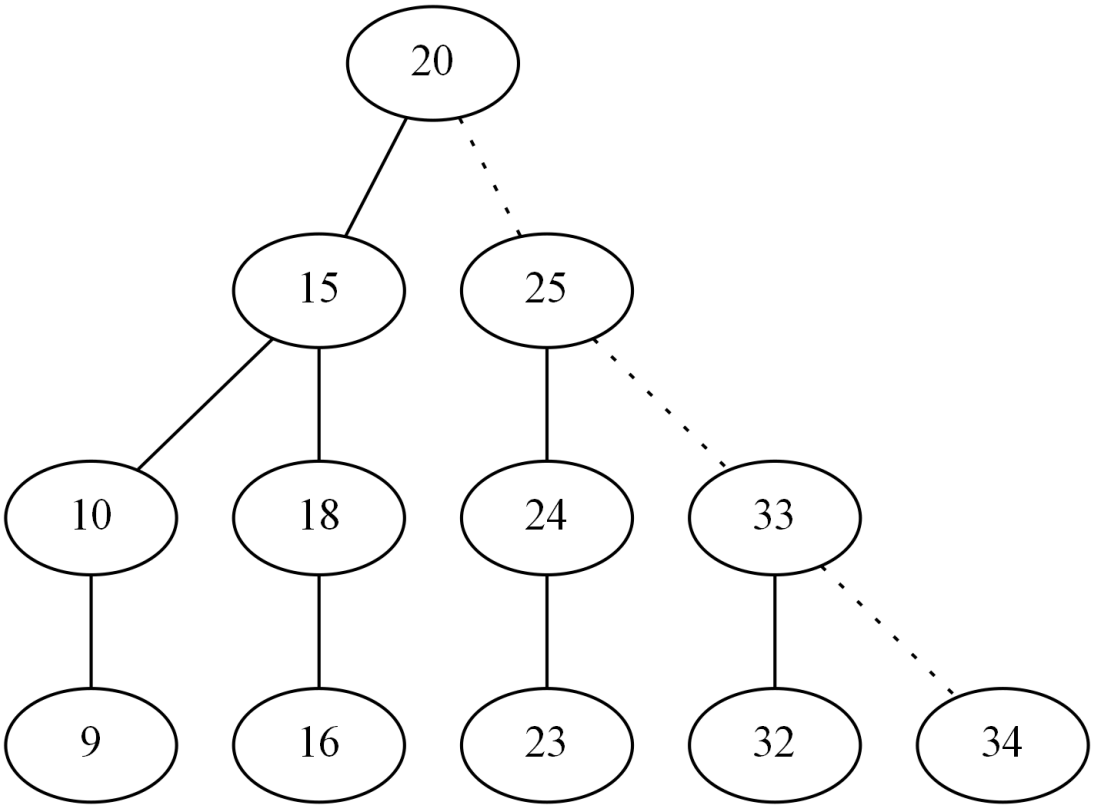

Рис. 1: поиск значения 34.

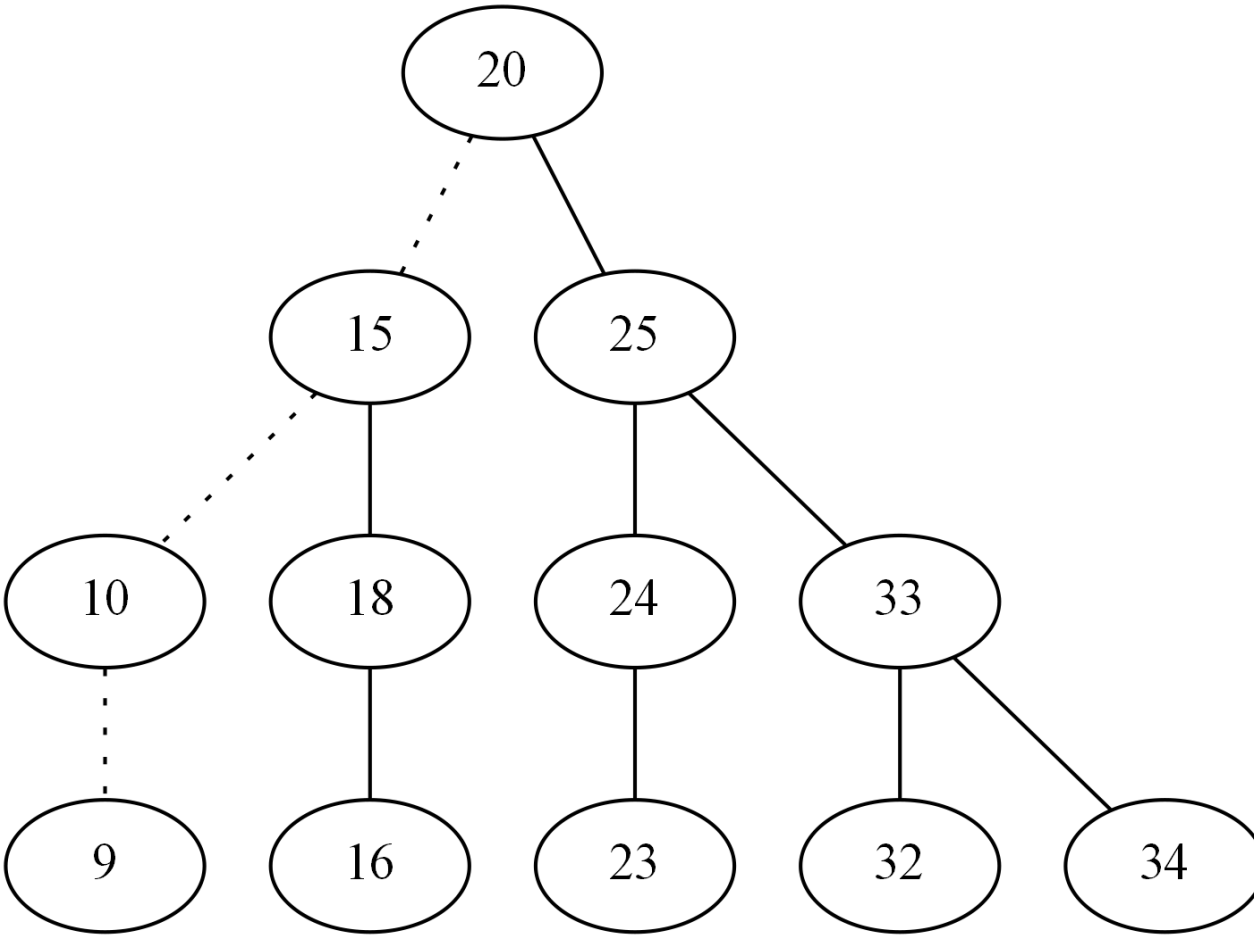

Рис. 2: поиск значения 9.

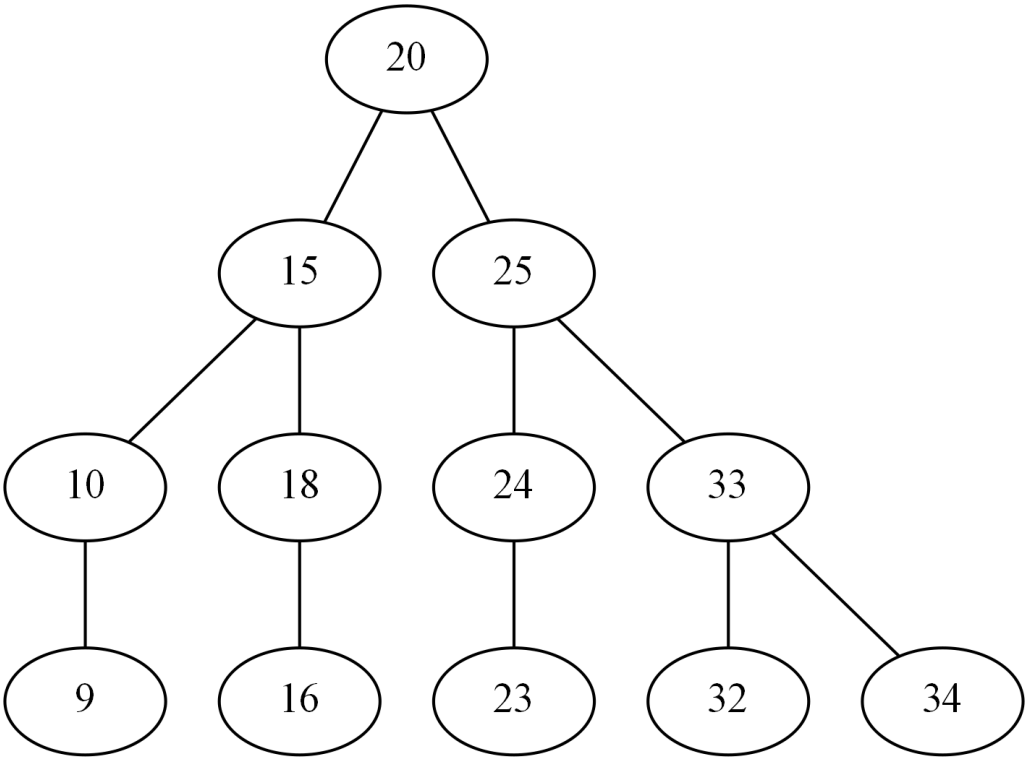

Рис. 3: поиск значения 100.

2. ВСТАВКА УЗЛА В ДЕРЕВО

2.1. Функция *insert(root, value)*

Для вставки элемента в дерево была реализована функция *insert(self, root, key)*, которой в качестве аргументов подается корень дерева и значение, которое необходимо вставить. Так как функция рекурсивная, то точка выхода - это проверка не является ли корень None. Если является, то мы создаем узел и возвращаем его. Далее в зависимости от значения мы движемся влево или вправо. После этого мы пересчитываем высоту и проверяем баланс. В процессе добавления в АВЛ-дереве возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает разбалансировка поддерева. Для выправления ситуации применяются повороты вокруг тех или иных узлов дерева.

2.2. Демонстрация поворотов

Демонстрация поворотов:

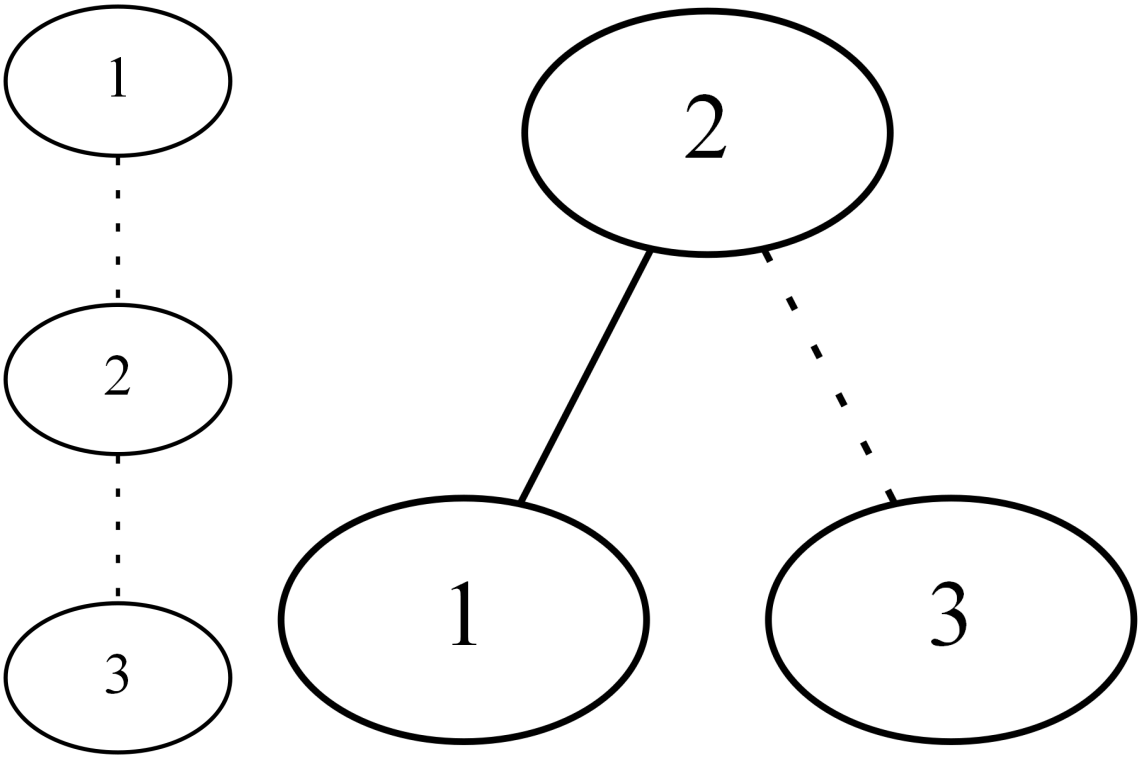

Рис. 4: Демонстрация малого правого вращения.

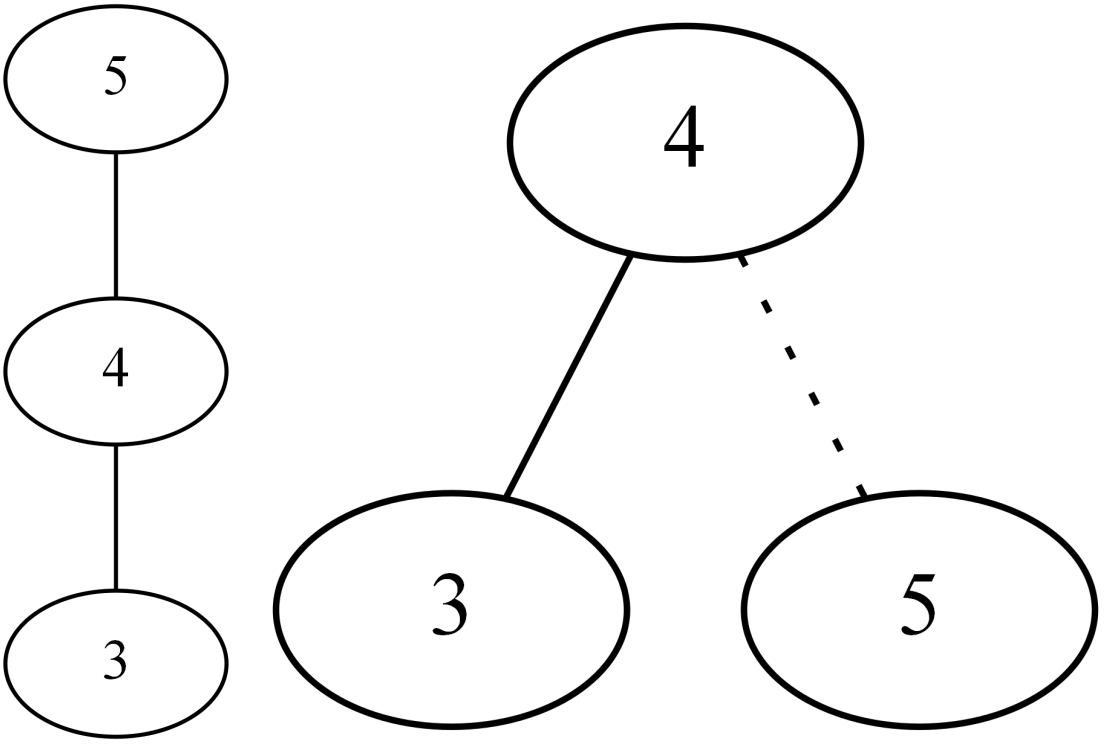

Рис. 5: Демонстрация малого левого вращения

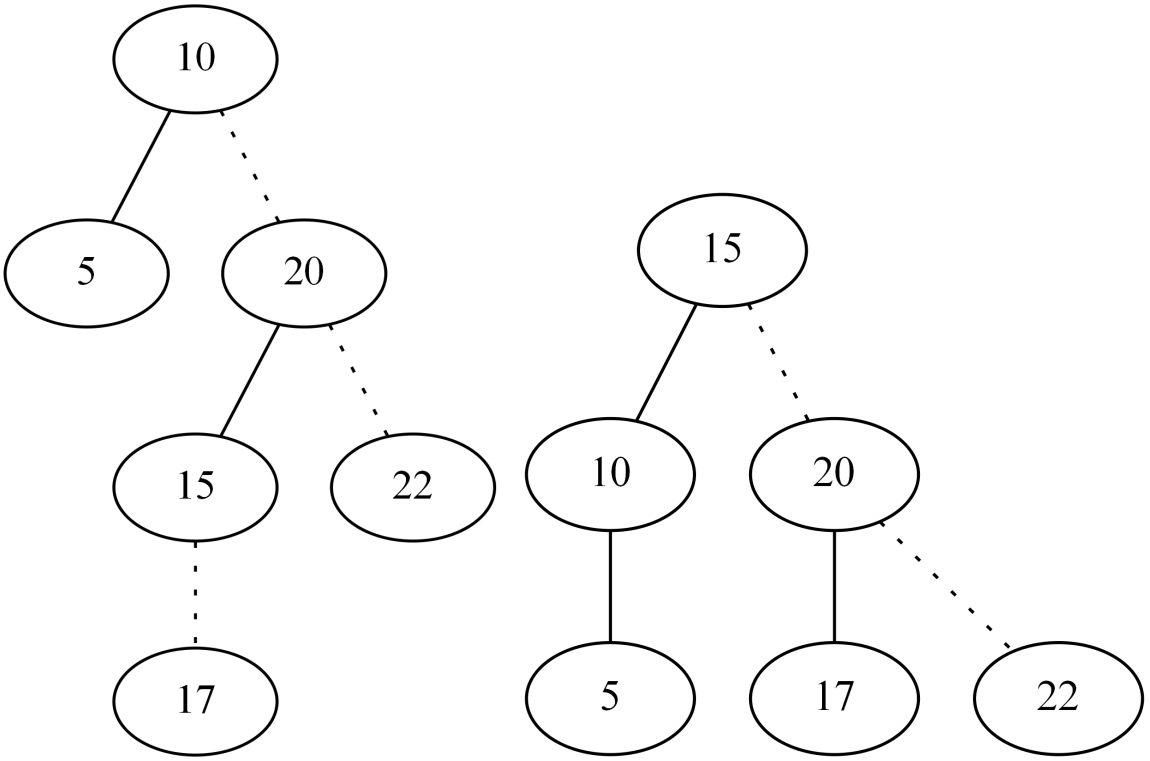

Рис. 6: Демонстрация большого левого вращения

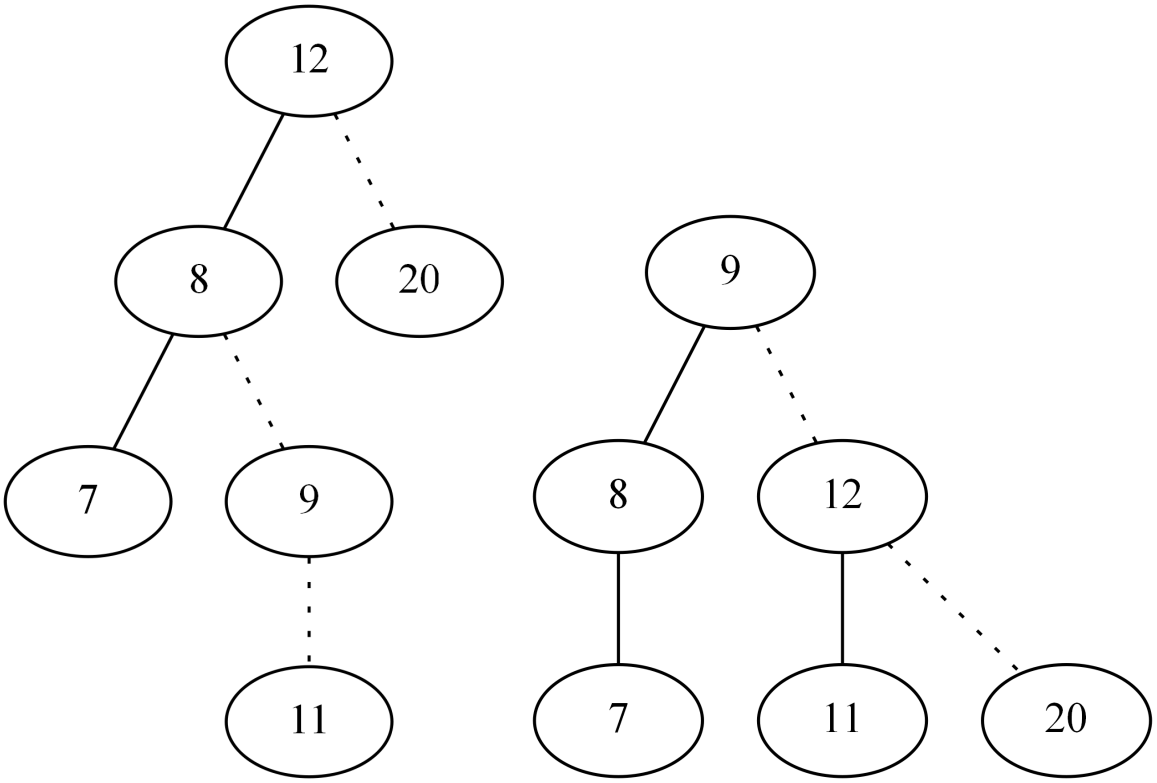

Рис. 7: Демонстраци большого правого вращения

3. ТЕСТИРОВАНИЕ

3.1. Тестирование

Результаты тестирования представлены в табл. 1.

Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	Проверка работы малого левого и малого правого вращения	-	Работает корректно
2.	Проверка работы большого левого и большого правого вращения	-	Работает корректно
3.	Поиск узла, который находится в дереве	-	Работает корректно
4.	Поиск узла, которого нет в дереве	-	Работает корректно

ЗАКЛЮЧЕНИЕ

Был изучен механизм работы AVL-деревьев, разработана функции вставки и поиска вершин и продемонстрирована их работа с помощью библиотеки pydot.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Habr URL: https://habr.com/ru/post/150732 (дата обращения: 07.12.2021).

2. Favtutor URL: https://favtutor.com/blogs/avl-tree-python (дата обращения: 07.12.2021)

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
from main_logic import AVLTree


if __name__ == '__main__':
    Tree = AVLTree()
    root = None
    while True:
        try:
            print("Type 'Exit' to exit the program.")
            print("Type 'Search' to search a value.")
            print("Type 'Show' to show the tree.")
            print("What to insert in AVL-tre? Data: ", end='')
            data = input()
            if data == 'Exit':
                break
            if data == 'Search':
                root_found = int(input())
                root_found = Tree.search(root, root_found)
                Tree.breadth_first_search(root, 'search.png')
                break
            if data == 'Show':
                Tree.breadth_first_search(root, 'show.png')
                break
            root = Tree.insert(root, int(data))
        except (TypeError, ValueError, AttributeError):
            print("Smth went wrong, plz try again!")
            print("Probably, you typed not a number.")
            print()
```

Название файла: main_logic.py

```
import pydot


class Node(object):
    def __init__(self, value):
        self.value = value
        self.left = None
        self.right = None
        self.h = 1  # высота считается в вершинах (в ребрах - 0)

    def __str__(self):
        left = self.left.value if self.left else None
        right = self.right.value if self.right else None
          return 'key: {}, left: {}, right: {}'.format(self.value,
left, right)
```

```
class AVLTree(object):
    def __init__(self):
        self.style_left = 'line'
        self.style_right = 'dotted'
        self.visited = []

    def insert(self, root, key, show_left=True, show_right=True):
        if not root:
            return Node(key)
        elif key < root.value:
            root.left = self.insert(root.left, key)
        else:
            root.right = self.insert(root.right, key)
        root.h = 1 + max(self.getHeight(root.left),
                         self.getHeight(root.right))

        b = self.getBal(root)

        if show_right:
            # Малое правое вращение
            if b > 1 and key < root.left.value:
                return self.rRotate(root)

        if show_left:
            # Малое левое вращение
            if b < -1 and key > root.right.value:
                return self.lRotate(root)

        if show_left and show_right:
            # Большое правое вращение
            if b > 1 and key > root.left.value:
                root.left = self.lRotate(root.left)
                return self.rRotate(root)
            # Большое левое вращение
            if b < -1 and key < root.right.value:
                root.right = self.rRotate(root.right)
                return self.lRotate(root)

        return root

    def lRotate(self, node_a):  # z
        node_b = node_a.right
        T2 = node_b.left

        node_b.left = node_a
        node_a.right = T2

        node_a.h = 1 + max(self.getHeight(node_a.left),
                           self.getHeight(node_a.right))
        node_b.h = 1 + max(self.getHeight(node_b.left),
                           self.getHeight(node_b.right))
```

```
            return node_b

        def rRotate(self, node_a):

            node_b = node_a.left
            T3 = node_b.right

            node_b.right = node_a
            node_a.left = T3

            node_a.h = 1 + max(self.getHeight(node_a.left),
                               self.getHeight(node_a.right))
            node_b.h = 1 + max(self.getHeight(node_b.left),
                               self.getHeight(node_b.right))

            return node_b

        def getHeight(self, root):
            if not root:
                return 0

            return root.h

        def getBal(self, root):
            if not root:
                return 0
            # if root.left is not None and root.right is not None:
                      #           return  self.getHeight(root.left)  -
self.getHeight(root.right)
            # elif root.left is not None and root.right is None:
            #     return self.getHeight(root.left)
            # elif root.left is None and root.right is not None:
            #     return -self.getHeight(root.right)
                               return   self.getHeight(root.left)   -
self.getHeight(root.right)

        def preOrder(self, root):

            if not root:
                return

            print("{0} ".format(root.value), end="")
            self.preOrder(root.left)
            self.preOrder(root.right)

        def search(self, root, value):
            if root is None:
                self.visited = []
                return
                # return False
            if root.value == value:
                return root
                # return True
```

```
        elif value < root.value:
            root = root.left if root.left is not None else None
            self.visited.append(root)
            root = self.search(root, value)
        else:
            root = root.right if root.right is not None else None
            self.visited.append(root)
            root = self.search(root, value)
        return root

    def inpre(self, root):
        while root.right is not None:
            root = root.right
        return root

    def insuc(self, root):
        while root.left is not None:
            root = root.left
        return root

    def remove(self, root, value):
        if root.left is None and root.right is None:
            root = None
            return None
        if root.value < value:
            root.right = self.remove(root.right, value)
        elif root.value > value:
            root.left = self.remove(root.left, value)
        else:
            if root.left is not None:
                q = self.inpre(root.left)
                root.value = q.value
                root.left = self.remove(root.left, q.value)
            else:
                q = self.insuc(root.right)
                root.value = q.value
                root.right = self.remove(root.right, q.value)
        if root is None:
            return root
            # update the height of the tree
        root.height = 1 + max(self.getHeight(root.left),
                              self.getHeight(root.right))

        balance = self.getBal(root)
        # Left Left
        if balance > 1 and self.getBal(root.left) >= 0:
            return self.rRotate(root)
        # Right Right
        if balance < -1 and self.getBal(root.right) <= 0:
            return self.lRotate(root)
        # Left Right
        if balance > 1 and self.getBal(root.left) < 0:
            root.left = self.lRotate(root.left)
```

```
                return self.rRotate(root)
            # Right Left
            if balance < -1 and self.getBal(root.right) < 0:
                root.right = self.rRotate(root.right)
                return self.lRotate(root)

            return root

        def checking(self):
            if len(self.visited) > 0:
                self.style_left = 'line'
                self.style_right = 'line'
            else:
                self.style_left = 'line'
                self.style_right = 'dotted'

        def breadth_first_search(self, root, name):
            graph = pydot.Dot(graph_type="graph")
            graph.obj_dict['attributes']["size"] = '"10,10!"'
            graph.obj_dict['attributes']["dpi"] = "144"
            graph.add_node(pydot.Node(root.value))
            self.checking()
            vertices_count = 0
            queue = [root]
            while queue:
                tmp_queue = []
                for element in queue:
                    if element.left:
                        if len(self.visited) > vertices_count:
                                    if self.visited[vertices_count] ==
element.left:
                                self.style_left = 'dotted'
                                vertices_count += 1
                        graph.add_node(pydot.Node(element.left.value))
                              graph.add_edge(pydot.Edge(element.value,
element.left.value, style=self.style_left))
                        tmp_queue.append(element.left)
                    if element.right:
                        if len(self.visited) > vertices_count:
                                    if self.visited[vertices_count] ==
element.right:
                                self.style_right = 'dotted'
                                vertices_count += 1

graph.add_node(pydot.Node(element.right.value))
                              graph.add_edge(pydot.Edge(element.value,
element.right.value, style=self.style_right))
                        tmp_queue.append(element.right)
                    # style_right = 'line'
                    # style_left = 'line'
                    self.checking()
                    queue = tmp_queue
            graph.write(name, format='png')
```

```
        self.visited = []
```