

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

**КУРСОВАЯ РАБОТА**  
**по дисциплине «Алгоритмы и структуры данных»**  
**Тема: AVL-деревья — вставка и исключение.**

Студент гр. 9303

\_\_\_\_\_

Алексеев Б.

Преподаватель

\_\_\_\_\_

Филатов Ар. Ю.

Санкт-Петербург

2020

## **ЗАДАНИЕ**

### **НА КУРСОВУЮ РАБОТУ (КУРСОВОЙ ПРОЕКТ)**

Студент Алексеенко Б.

Группа 9303

Тема работы (проекта): АВЛ-деревья — вставка и исключение.

Исследование(в среднем и в худшем случае).

Исходные данные:

произвольные наборы чисел.

Содержание пояснительной записки:

Перечисляются требуемые разделы пояснительной записки (обязательны разделы «Содержание», «Введение», «Заключение», «Список использованных источников»)

Предполагаемый объем пояснительной записки:

Не менее 10 страниц.

Дата выдачи задания: 6.11.2020

Дата сдачи реферата: 25.12.2020

Дата защиты реферата: 25.12.2020

Студент

\_\_\_\_\_

Алексеенко Б.

Преподаватель

\_\_\_\_\_

Филатов Ар. Ю.

## **АННОТАЦИЯ**

В данной работе было проведено исследование, в котором были оценены и рассмотрены худшие и средние случаи вставки и удаления элементов в сбалансированное бинарное АВЛ-дерево. Во время работы были построены графики с помощью написанных программ, на которых явно видно подтверждение теоретических предположений. Была реализована структура классов, которая представляет собой АВЛ-дерево, а так же код генерирующий наборы случайных чисел для демонстрации среднего и худшего случаев вставки и удаления в АВЛ-дерево.

## **SUMMARY**

In this paper, a study was conducted in which the worst and average cases of insertion and removal of elements in a balanced binary AVL tree were evaluated and considered. During the work, graphs were built using written programs, which clearly show the confirmation of theoretical assumptions. A class structure was implemented, which is an AVL tree, as well as a code that generates sets of random numbers to demonstrate the average and worst cases of insertion and deletion into an AVL tree.

## СОДЕРЖАНИЕ

Введение	4
1. Описание AVL-дерева.	5
1.1. Общие сведения	6
1.2. Реализация	6
2. Исследование	9
2.1. Теоретическая оценка сложности	9
2.2. Генерация случайного набора входных данных	10
2.3. Практическая оценка алгоритмов	10
Заключение	15
Приложение А. Исходный код программы.	17
Приложение Б. Исходный код вспомогательных программ	32

## ВВЕДЕНИЕ

Цели исследования:

- Реализовать структуру классов, представляющую AVL-дерево.
- Провести анализ большого объема данных и построить графики.

Исследование проводилось со случайно сгенерированными наборами данных.

План экспериментального исследования:

Реализовать структуру классов, которая будет представлять AVL-дерево.

Реализовать программу, генерирующую наборы случайных чисел, для тестирования вставки и удаления элементов.

- Выбрать базовые операции и реализовать автоматический подсчет выбранных операций при вставке и удалении элемента в AVL-дерево.
- С помощью программы построить графики зависимости количества элементов к количеству базовых операций.
- Сделать выводы по полученным графикам и оценить сложность алгоритмов, сопоставить ее с теоретической.

# 1. ОПИСАНИЕ АВЛ-ДЕРЕВА

## 1.1. Общие сведения

АВЛ-дерево — это двоичное дерево поиска, ключи которого удовлетворяют стандартному свойству: ключ любого узла дерева не меньше любого ключа в левом поддереве данного узла и не больше любого ключа в правом поддереве этого узла.

Особенностью АВЛ-дерева является то, что оно является сбалансированным в следующем смысле: *для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу*. Доказано, что этого свойства достаточно для того, чтобы высота дерева логарифмически зависела от количества его узлов. А так как основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, то получаем *гарантированную* логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве.

Традиционно, узлы АВЛ-дерева хранят не высоту, а разницу высот правого и левого поддеревьев (так называемый *balance factor*), которая может принимать только три значения -1, 0 и 1. Вспомним, что высота  $h < 1.44 \log_2(n + 2)$ , это значит, например, что при  $n = 10^9$  высота дерева не превысит величины  $h = 44$  (!).

## 1.2. Реализация

Класс `Node_AVL_Tree` представляет собой узел АВЛ-дерева, в узле хранятся указатели на левый и правый элемент, значение высоты поддерева, разницу высот левого и правого поддерева, а так же булеву переменную `rotate`, которая показывает, был ли произведен какой-либо поворот данного узла.

Класс Head\_AVL\_Tree представляет собой голову AVL-дерева, в нем хранится указатель на корень AVL-дерева, а так же расчетное значение  $O(n)$  последней выполненной операции.

Методы класса Head\_AVL\_Tree:

void insert(Type) — вставка указанного элемента в дерево.

void print\_tree(Node\_AVL\_Tree<Type>\*, int level) — вывод построенного дерева в консоль.

void is\_contain(Type) — возвращает истинну, если заданный элемент есть в дереве.

Void remove(Type) — удаляет указанный элемент из AVL-дерева.

Node\_AVL\_Tree<Type>\* get\_head() — метод , который возвращает указатель на корень дерева.

Int get\_o() - метод, который возвращает расчетное значение  $O(n)$  последней исполненной операции.

Методы класса Node\_AVL\_Tree:

Node\_AVL\_Tree<Type>\* - метод, который осуществляет вставку заданного элемента и, если это необходимо, производит балансировку дерева.

Node\_AVL\_Tree<Type>\* remove — метод, который осуществляет удаление элемента и, если это необходимо, производит балансировку дерева.

Node\_AVL\_Tree<Type>\* remove\_min — метод, который удаляет минимальный элемент из узла и, если это необходимо, переставивает поддереву.

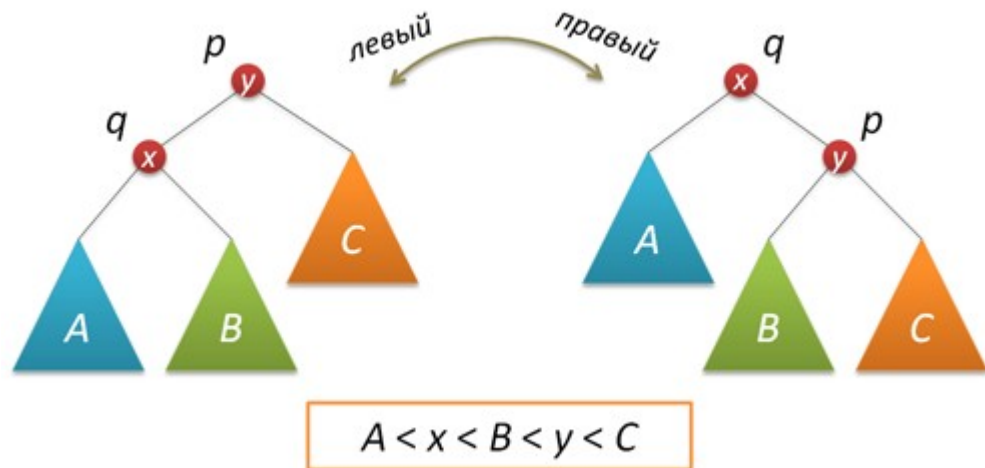
Node\_AVL\_Tree<Type>\* find\_min — метод, который находит минимум в AVL-дерева.

Node\_AVL\_Tree<Type>\* left\_rotate — метод, который производит поворот узла влево.

Node\_AVL\_Tree<Type>\* right\_rotate — метод, который производит правый поворот узла.

Node\_AVL\_Tree<Type>\* make\_balance — метод, который производит балансировку дерева с помощью четырех видов поворотов.

В процессе добавления или удаления узлов в AVL-дереве возможно возникновение ситуации, когда `balance` некоторых узлов оказывается равными 2 или -2, т.е. возникает *расбалансировка* поддерева. Для выправления ситуации повороты вокруг тех или иных узлов дерева. На картинке иллюстрирован правый поворот, левый будет производиться зеркально.



Методы `get_r()` и `get_l()` возвращают указатели на, соответственно, правое и левое поддерево. Метод `get_d()` возвращает значение елюча в данном дереве.

Функция `countDeer` производит расчет высоты узлов поддеревьев.



## 2. ИСЛЕДОВАНИЕ

### 2.1. Теоретическая оценка сложности.

#### Вставка

При вставке некоторого ключа мы будем спускаться по дереву. Если мы находимся в узле  $p$  и нам необходимо переместиться в поддереву, которого не существует, то преобразуем ключ в лист, а сам узел делаем его предком. Поднимаемся вверх по пути поиска и пересчитываем  $balance$ . Если поднялись в узел  $q$  из правого поддерева, то увеличиваем  $balance$  на единицу, а если из левого — уменьшаем на единицу. Если  $balance$  в узле равен 0, то высота поддерева не изменилась и можно прекратить подъем. Если же  $balance$  равен 1 или -1, то высота дерева явно изменилась и подъем необходимо продолжить. Если мы находимся в узле, где  $balance$  2 или -2, тогда необходимо произвести большой или малый поворот влево или вправо (в зависимости от остального дерева), если после этого  $balance$  будет равен 0 — заканчиваем подъем.

Т. е. в процессе вставки мы проходим по не более чем  $h$  узлов дерева, а для каждого узла запускаем балансировку не более одного раза, тогда количество всех операций при вставке элемента в AVL-дерево равно  $O(\log(n))$ , где  $n$  — количество элементов.

#### Удаление

В том случае, если у удаляемого узла нет поддеревьев, то просто удалим его, в других же случаях перейдем к самому левому узлу правого поддерева, переместим его на место удаляемого, после чего удалим, собственно, удаляемый. От удаленной вершины теперь необходимо вернуться к корню и произвести перерасчет  $balance$  во всех узлах. Если находясь в некотором узле, поднявшись в него из правого поддерева, то уменьшим его  $balance$  на единицу. Если  $balance$  узле равен 0, тогда высота поддерева не изменилась, а если же  $balance$  равен 1 или -1, тогда продолжаем подъем. В том случае если  $balance$  будет меньше -1 или больше 1, то произведем возврат баланса.

Тогда при удалении элемента мы так же пройдем не более чем  $h$  узлов и количество операций, затраченных на удаление, будет равно, по свойству АВЛ-дерева,  $O(\log(n))$ .

Операция	В среднем случае	В худшем случае
Поиск	$O(\log(n))$	$O(\log(n))$
Вставка	$O(\log(n))$	$O(\log(n))$
Удаление	$O(\log(n))$	$O(\log(n))$

## 2.2. Генерация случайного набора чисел

Для подтверждения теоретической оценки были реализованы функции, которые генерируют случайные наборы чисел, только один из них — возрастающая последовательность, а второй состоит из случайных чисел. Сами деревья генерируются тоже случайным образом

## 2.3. Практическая оценка алгоритма.

Для каждого случая генерируется набор из 10 тысяч значений, после чего расчетное  $O(n)$  каждой операции записывается в файл. В итоге получается 200 файлов для рассмотрения одного случая, т. е., т. к. рассматривается всего 4 случая, получаем 1600 файлов для анализа. Так же программа была протестирована на генерацию деревьев с 100 тысячами значений, показав себя отлично.



Рисунок 1-Файлы, сгенерированные для среднего случая удаления

На основе вычислений были построены графики, для этого был использован код, написанный на языке Python 3. На всех графиках оранжевым цветом отображается функция  $\log(n)$ . Синяя кривая — количество вызовов базовых операций для определенного количества операций.

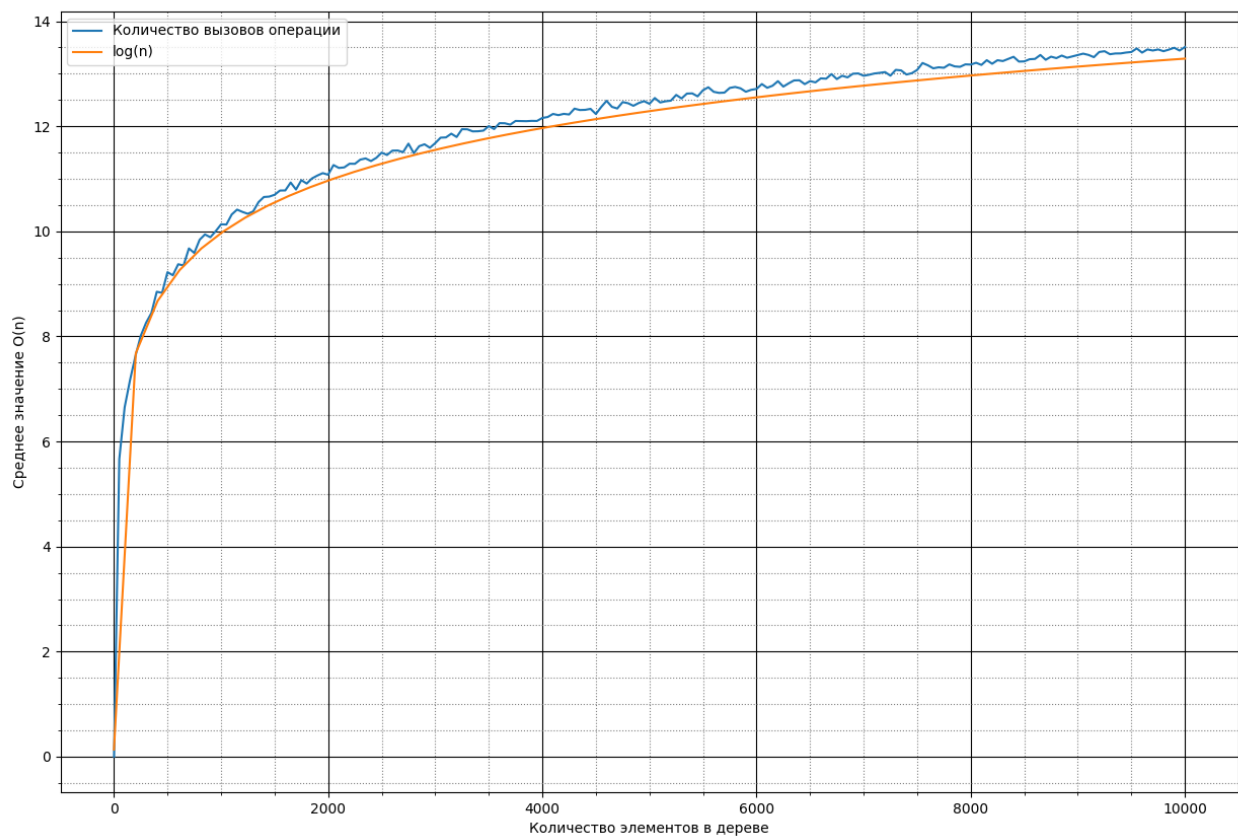


Рисунок 2-Удаление, средний случай

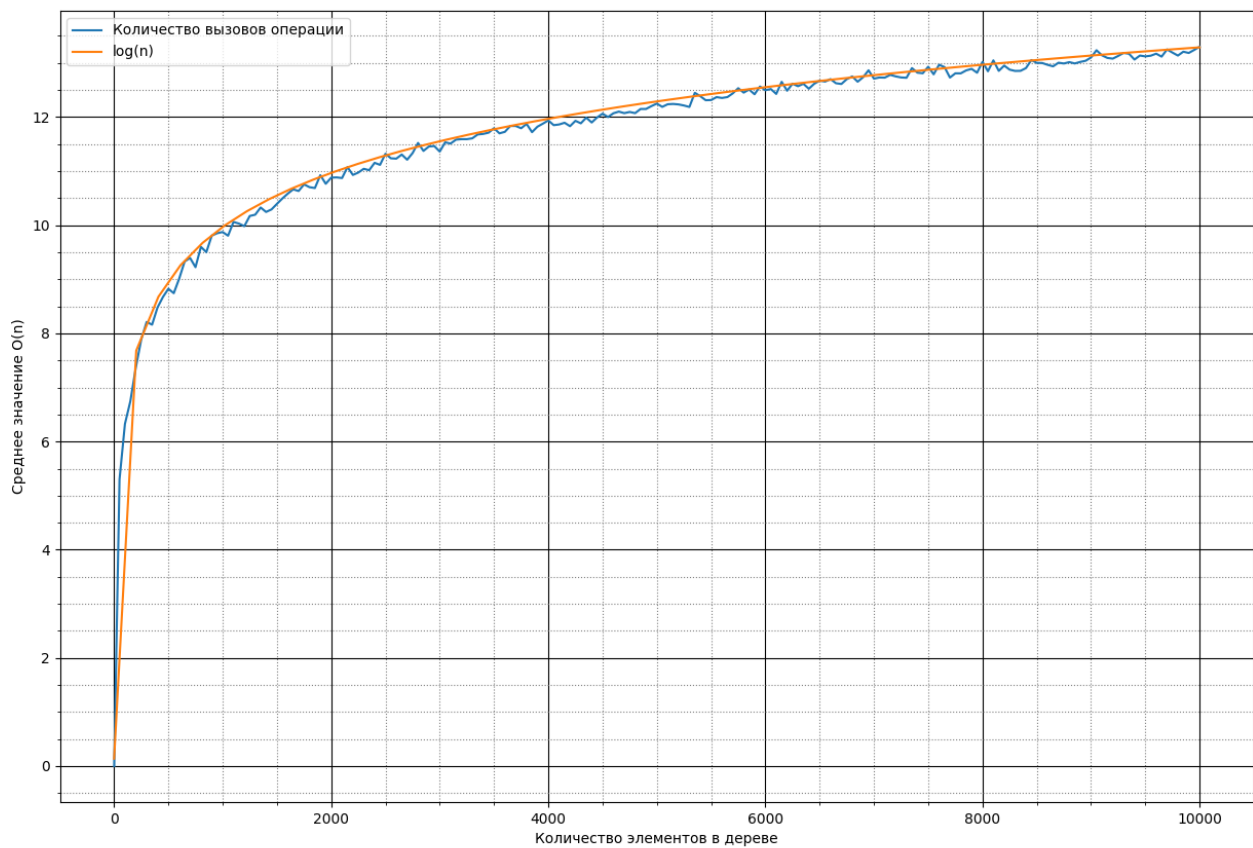


Рисунок 3-Удаление, худший случай

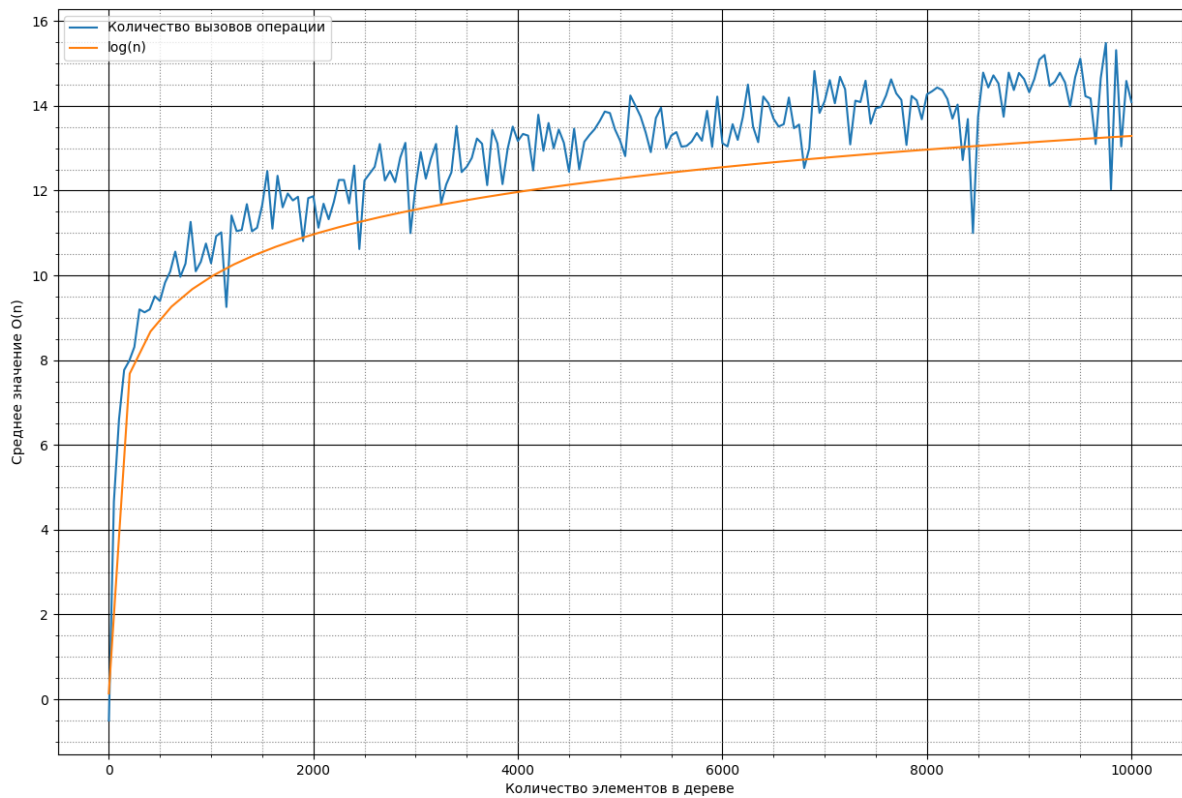


Рисунок 4-Вставка, средний случай

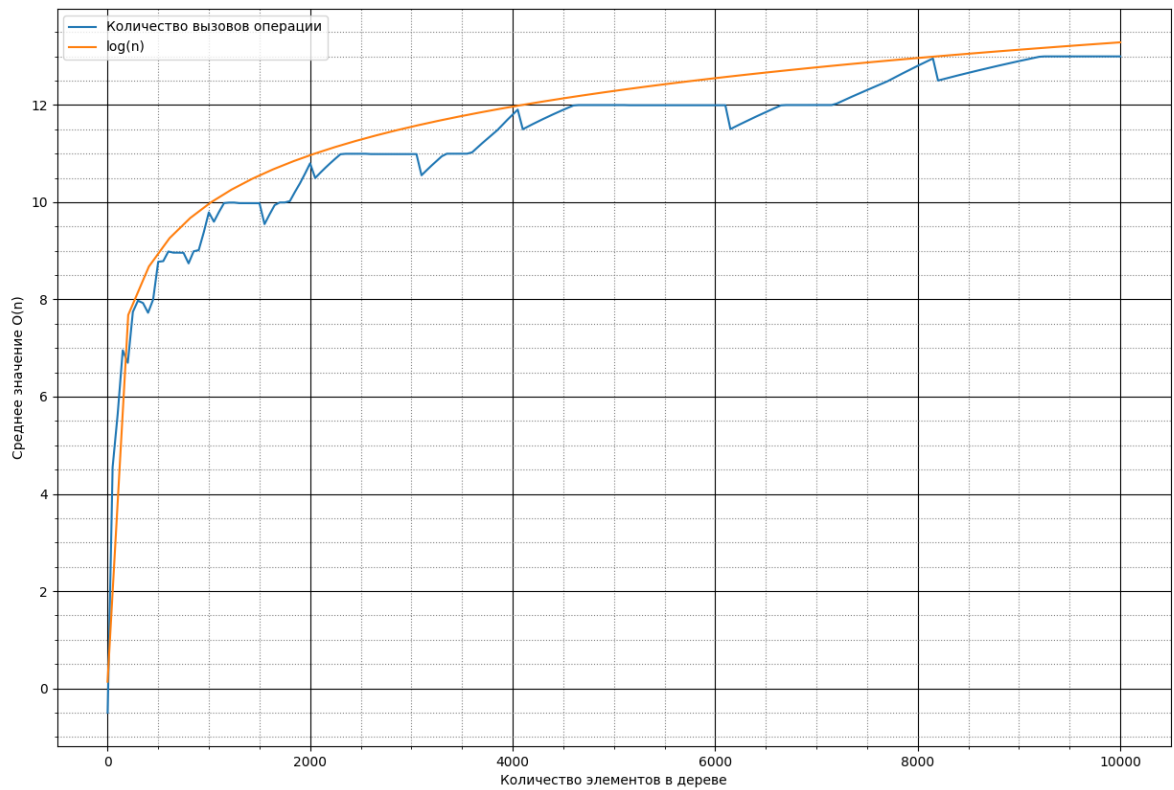


Рисунок 5-Вставка, худший случай

На графиках видно, как зависит количество вызовов определенных функций от количества элементов в дереве. По графику явно видно, что сложность алгоритма составляет примерно  $\log(n)$ , что подтверждает теоретическую оценку сложности вставки и удаления в AVL-деревьях.

## **ЗАКЛЮЧЕНИЕ**

В ходе работы была реализована структура классов, которая представляет из себя AVL-дерево, были реализованы методы вставки и удаления значений. На основе полученных числовых значений были построены графики.

Полученные практические результаты были сравнены с теоретическими и подтвердили их. Таким образом, была доказана теоретическая оценка асимптотики работы операций вставки и удаления в AVL-дерево.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. AVL-дерево  
<https://ru.wikipedia.org/wiki/%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>
2. AVL-дерево  
<https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>
3. AVL-деревья  
<https://habr.com/ru/post/150732/>
4. Структуры данных. AVL дерево  
<https://medium.com/@dimko1/%D1%81%D1%82%D1%80%D1%83%D0%BA%D1%82%D1%83%D1%80%D1%8B-%D0%B4%D0%B0%D0%BD%D1%8B%D1%85-avl-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE-7f8739e8faf9>



## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: main.cpp

```
#include "AVL_tree.h"
#include <vector>
#include <ctime>
#include <cstdlib>
#include <algorithm>

using namespace std;

vector<int> avl_insert_generator(int size, char c){
    vector<int> avl_tree;
    if (c == 'w') {
        int m = rand();
        for (int i = m; i < m + size; i++) {
            avl_tree.push_back(i);
        }
    }
    else if (c == 'a') {
        for (int i = 0; i < size; i++) {
            avl_tree.push_back(rand());
        }
    }
    return avl_tree;
}

void avl_tree_generator_average(Head_AVL_Tree<int>* head,
vector<int>& the_ones, vector<int>& avl_tree, int size) {
    for (int i = 0; i < size; i++)
    {
        int elem = rand();
        head->insert(elem);
        avl_tree.push_back(elem);
    }

    std::cout << avl_tree.size() << endl;
    for (int i = 0; i < size / 2; i++)
    {
        int itr = rand() % avl_tree.size();
        std::cout << "Iter - " << itr << endl;
        the_ones.push_back(avl_tree[itr]);
    }
}

void avl_tree_generator_worst(Head_AVL_Tree<int>* head,
vector<int>& the_ones, vector<int>& avl_tree, int size) {
    int m = rand() % size;
```

```

        for (int i = m; i < size + m; i++) {
            head->insert(i);
            avl_tree.push_back(i);
        }

        std::cout << avl_tree.size() << endl;
        for (int i = 0; i < size / 2; i++)
        {
            int itr = rand() % avl_tree.size();
            if (itr > avl_tree.size()) {
                i--;
                continue;
            }

            std::cout << "Iter - " << itr << endl;
            the_ones.push_back(avl_tree[itr]);
        }
    }

    int main(int argc, char* argv[]) {

        srand(static_cast<unsigned int>(time(0)));

        std::cout << "Which operation you want to research? [Delete -
d / Insert - i]" << endl;

        char operation = 'd';
        if(argc > 1)
            operation = *argv[1];
        //cin >> operation;

        if (operation == 'd')
        {
            std::cout << "What case do you want to consider?[w -
worst / a - average]" << endl;
            char choice = 'w';

            if(argc > 2)
                choice = *argv[2];
            //cin >> choice;

            if (choice == 'w') {

                int size = 1;
                int iter = 0;

                while (size <= 10000)
                {
                    ofstream fout;
                    std::string file = "result_d_w/result_w_";

```

```

        std::string num = to_string(iter);
        file.append(num);
        file.append(".txt");
        fout.open(file, std::fstream::trunc |
std::fstream::out);

        Head_AVL_Tree<int>* head = new
Head_AVL_Tree<int>;
        vector<int> the_ones;
        vector<int> avl_tree;
        avl_tree_generator_worst(head, the_ones,
avl_tree, size);

        fout << size << " ";

        for (int i = 0; i < the_ones.size(); i++)
        {
            if (head->is_contain(the_ones[i])) {
                if (i != 0) {
                    fout << " ";
                }
                head->reset_o();
                head->remove(the_ones[i]);
                fout << head->get_o() - 2;
                head->insert(rand() % size);
            }
        }

        fout << endl;
        fout << endl;

        if (size == 1) {
            size += 49;
        }
        else
        {
            size += 50;
        }
        iter++;
        delete head;
        fout.close();
    }

    else if (choice == 'a') {
        int size = 1;
        int iter = 0;

        while (size <= 10000)
        {
            ofstream fout;
            std::string file = "result_d_a/result_a_";

```

```

        std::string num = to_string(iter);
        file.append(num);
        file.append(".txt");
        fout.open(file, std::fstream::trunc |
std::fstream::out);

        Head_AVL_Tree<int>* head = new
Head_AVL_Tree<int>;

        vector<int> the_ones;
        vector<int> avl_tree;

        avl_tree_generator_average(head, the_ones,
avl_tree, size);

        fout << size << " ";

        for (int i = 0; i < the_ones.size(); i++)
        {
            if (head->is_contain(the_ones[i])) {
                if (i != 0) {
                    fout << " ";
                }
                head->reset_o();
                head->remove(the_ones[i]);
                fout << head->get_o() - 2;
                head->insert(rand());
            }
        }
        fout << endl;
        fout << endl;

        if (size == 1) {
            size += 49;
        }
        else
        {
            size += 50;
        }

        iter++;
        delete head;
        fout.close();
    }
}

else if(operation == 'i'){
    std::cout << "What case do you want to consider?[w -
worst / a - average]" << endl;
    //cin >> choice;
    char choice = 'w';
    if(argc > 2)
        choice = *argv[2];

```

```

        if (choice == 'w') {

            int size = 1;
            int iter = 0;

            while (size <= 10000)
            {
                ofstream fout;
                std::string file = "result_i_w/result_w_";
                std::string num = to_string(iter);
                file.append(num);
                file.append(".txt");
                fout.open(file, std::fstream::trunc |
std::fstream::out);

                Head_AVL_Tree<int>* head = new
Head_AVL_Tree<int>;
                vector<int> avl_tree;
                vector<int> the_ones;
                avl_tree_generator_worst(head, the_ones,
avl_tree, size);
                avl_tree = avl_insert_generator(size, 'w');

                fout << size << " ";

                for (int i = 0; i < avl_tree.size(); i++)
                {
                    if (i != 0) {
                        fout << " ";
                    }
                    head->reset_o();
                    head->insert(avl_tree[i]);
                    fout << head->get_o() - 2;
                    head->remove(head->get_head()-
>get_d());
                }

                fout << endl;
                fout << endl;

                if (size == 1) {
                    size += 49;
                }
                else
                {
                    size += 50;
                }
                iter++;
                delete head;
                fout.close();
            }
        }
    }
}

```

```

else if (choice == 'a') {
    int size = 1;
    int iter = 0;

    while (size <= 10000)
    {
        ofstream fout;
        std::string file = "result_i_a/result_a_";
        std::string num = to_string(iter);
        file.append(num);
        file.append(".txt");
        fout.open(file, std::fstream::trunc |
std::fstream::out);

        Head_AVL_Tree<int>* head = new
Head_AVL_Tree<int>;

        vector<int> the_ones;
        vector<int> avl_tree;

        avl_tree_generator_average(head, the_ones,
avl_tree, size);

        avl_tree = avl_insert_generator(size, 'w');

        fout << size << " ";

        for (int i = 0; i < avl_tree.size(); i++)
        {
            if (i != 0) {
                fout << " ";
            }
            head->reset_o();
            head->insert(avl_tree[i]);
            fout << head->get_o() - 2;
            head->remove(head->get_head()->get_d());
        }

        if (size == 1) {
            size += 49;
        }
        else
        {
            size += 50;
        }

        iter++;
        delete head;
        fout.close();
    }
}

```

```

        std::cout << "Finished right" << endl;
        return 0;
}

```

Имя файла: AVL\_Tree.h

```

#ifndef AVL_TREE_H
#define AVL_TREE_H
#include <iostream>
#include <fstream>
#include <string>
#include <map>
#include <iostream>
#include <fstream>
#include <stack>

template <class Type>
class Head_AVL_Tree;

template <class Type>
class Node_AVL_Tree {
public:
    friend class Head_AVL_Tree<Type>;

    bool is_contain(Type, int);
    int set_height();
    int set_balance();

    class Node_AVL_Tree<Type>* insert(Type, int& count_o);
    class Node_AVL_Tree<Type>* remove(Type, int& count_o);
    class Node_AVL_Tree<Type>* remove_min(int& count_o);
    class Node_AVL_Tree<Type>* find_min(int& count_o);
    class Node_AVL_Tree<Type>* left_rotate(int& count_o);
    class Node_AVL_Tree<Type>* right_rotate(int& count_o);
    class Node_AVL_Tree<Type>* make_balance(int& count_o);
    class Node_AVL_Tree<Type>* get_r();
    class Node_AVL_Tree<Type>* get_l();
    Type get_d();
    void set_rot(bool);
    bool get_rot();
    int countDeep(Node_AVL_Tree<Type>* node);
    Node_AVL_Tree();
    ~Node_AVL_Tree();

private:
    bool rotate = false;
    int height;
    int balance;
    Type data;
    class Node_AVL_Tree<Type>* left;

```

```

        class Node_AVL_Tree<Type>* right;
};

template <class Type>
class Head_AVL_Tree {
public:
    Head_AVL_Tree();
    ~Head_AVL_Tree();
    void insert(Type);
    void print_tree(Node_AVL_Tree<Type>* node, int level);
    bool is_contain(Type);
    void remove(Type);
    Node_AVL_Tree<Type>* get_head();
    int get_o();
    void reset_o();
private:
    int count_o = 0;
    class Node_AVL_Tree<Type>* head;
};

```

```

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::get_r() {
    return right;
}

```

```

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::get_l() {
    return left;
}

```

```

template <class Type>
Type Node_AVL_Tree<Type>::get_d() {
    return data;
}

```

```

template <class Type>
void Node_AVL_Tree<Type>::set_rot(bool rot) {
    rotate = rot;
}

```

```

template <class Type>
bool Node_AVL_Tree<Type>::get_rot() {
    return rotate;
}

```

```

template <class Type>
Node_AVL_Tree<Type>::Node_AVL_Tree() {

```



```

        left = nullptr;
        right = nullptr;
    }

template <class Type>
Node_AVL_Tree<Type>::~~Node_AVL_Tree() { /
    if (left)
        delete left;
    if (right)
        delete right;
}

template <class Type>
int Node_AVL_Tree<Type>::countDeep(Node_AVL_Tree<Type>* node)
{
    if (node == nullptr)
        return 0;
    int cl = countDeep(node->get_l());
    int cr = countDeep(node->get_r());
    return 1 + ((cl > cr) ? cl : cr);
}

template <class Type>
bool Node_AVL_Tree<Type>::is_contain(Type desired, int depth) {
    if (data == desired)
        return true;
    if (left && data > desired) {
        std::cout << "find in left" << std::endl;
        if (left->is_contain(desired, depth + 1))
            return true;
    }
    if (right && data < desired) {
        std::cout << "find in right" << std::endl;
        return right->is_contain(desired, depth + 1);
    }
    return false;
}

template <class Type>
int countDeep(Node_AVL_Tree<Type>* node)
{
    if (node == nullptr)
        return 0;
    int cl = countDeep(node->get_l());
    int cr = countDeep(node->get_r());
    return 1 + ((cl > cr) ? cl : cr);
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::remove(Type
to_remove, int& count_o) {

```

```

count_o++;
if (data == to_remove) {
    if (!left && !right) {
        delete this;
        return nullptr;
    }
    if (!right) {
        class Node_AVL_Tree<Type>* temp = left;

        this->left = nullptr;
        delete this;
        return temp;
    }

    class Node_AVL_Tree<Type>* new_root;
    new_root = new Node_AVL_Tree<Type>;
    new_root->left = (right->find_min(count_o)->left);
    new_root->right = (right->find_min(count_o)->right);
    new_root->data = (right->find_min(count_o)->data);
    new_root->balance = (right->find_min(count_o)->balance);

    right = right->remove_min(count_o);
    new_root->left = left;
    new_root->right = right;
    new_root->height = set_height();
    new_root->balance = set_balance();
    return new_root->make_balance(count_o);
}
if (data < to_remove) {
    right = right->remove(to_remove, count_o);
}

if (data > to_remove) {
    left = left->remove(to_remove, count_o);
}
height = set_height();
balance = set_balance();
return make_balance(count_o);
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::find_min(int&
count_o) {
    //count_o++;
    return left ? left->find_min(count_o) : this;
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::remove_min(int&
count_o) {
    count_o++;
    if (!left) {
        class Node_AVL_Tree<Type>* temp = right;

```

```

        this->right = nullptr;
        delete this;
        return temp;
    }
    left = left->remove_min(count_o);
    count_o++;
    height = set_height();
    balance = set_balance();
    return make_balance(count_o);
}

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::insert(Type value,
int& count_o) {
    count_o++;
    if (value >= data) {
        if (!right) {
            right = new Node_AVL_Tree<Type>;
            right->data = value;
            right->height = 1;
        }
        else {
            right = right->insert(value, count_o);
        }
    }
    if (value < data) {
        if (!left) {
            left = new Node_AVL_Tree<Type>;
            left->data = value;
            left->height = 1;
        }
        else {
            left = left->insert(value, count_o);
        }
    }
    height = set_height();
    balance = set_balance();
    return make_balance(count_o);
}

```

```

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::right_rotate(int&
count_o) {
    count_o++;
    std::cout << "right rotate around element: " << this->data <<
std::endl;
    Node_AVL_Tree<Type>* temp;
    temp = left;

    temp->rotate = true;
    if (temp->left != nullptr)
        temp->left->rotate = true;

```

```

        if (temp->right != nullptr)
            temp->right->rotate = true;

        left = temp->right;
        this->height = this->set_height();
        this->balance = this->set_balance();
        if (temp->left) {
            temp->left->height = temp->left->set_height();
            temp->left->balance = temp->left->set_balance();
        }
        temp->right = this;
        temp->height = temp->set_height();
        temp->balance = temp->set_balance();
        return temp;
    }

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::left_rotate(int&
count_o) {
    count_o++;
    std::cout << "left rotate around element: " << this->data <<
std::endl;
    Node_AVL_Tree<Type>* temp;
    temp = right;

    temp->rotate = true;
    if (temp->left != nullptr)
        temp->left->rotate = true;
    if (temp->right != nullptr)
        temp->right->rotate = true;

    right = temp->left;
    this->height = this->set_height();
    this->balance = this->set_balance();
    if (temp->right) {
        temp->right->height = temp->right->set_height();
        temp->right->balance = temp->right->set_balance();
    }
    temp->left = this;
    temp->height = temp->set_height();
    temp->balance = temp->set_balance();
    return temp;
}

```

```

template <class Type>
class Node_AVL_Tree<Type>* Node_AVL_Tree<Type>::make_balance(int&
count_o) {
    Node_AVL_Tree<Type>* temp;
    temp = this;
    if (balance == 2) {
        if (right->balance == -1) {

```

```

        temp->right = right->right_rotate(count_o);
    }
    temp = left_rotate(count_o);
}
if (balance == -2) {
    if (left->balance == 1) {
        temp->left = left->left_rotate(count_o);
    }
    temp = right_rotate(count_o);
}
return temp;
}

template <class Type>
int Node_AVL_Tree<Type>::set_height() {
    if (!left && !right)
        return 1;
    if (!left)
        return (right->height + 1);
    if (!right)
        return (left->height + 1);
    if (left->height >= right->height)
        return (1 + left->height);
    if (left->height < right->height)
        return (1 + right->height);
    return 0;
}

template <class Type>
int Node_AVL_Tree<Type>::set_balance() {
    if (!left && !right)
        return 0;
    if (!left)
        return right->height;
    if (!right)
        return (left->height * (-1));
    return (right->height - left->height);
}

template <class Type>
Head_AVL_Tree<Type>::Head_AVL_Tree() {
    head = nullptr;
}

template <class Type>
Head_AVL_Tree<Type>::~~Head_AVL_Tree() {
    delete head;
}

template <class Type>
Node_AVL_Tree<Type>* Head_AVL_Tree<Type>::get_head() {
    return this->head;
}

```

```

template <class Type>
int Head_AVL_Tree<Type>::get_o() {
    return this->count_o;
}

template <class Type>
void Head_AVL_Tree<Type>::reset_o() {
    this->count_o = 0;
}

//----- print -----//

template <class Type>
void Head_AVL_Tree<Type>::print_tree(Node_AVL_Tree<Type>* node,
int level) {
    if (node)
    {
        print_tree(node->get_r(), level + 1);

        for (int i = 0; i < level; i++) {
            std::cout << "    ";
        }
        std::cout << node->get_d() << std::endl;

        print_tree(node->get_l(), level + 1);
    }
}

//----- print -----//

template <class Type>
void Head_AVL_Tree<Type>::insert(Type value) {
    if (!head) {
        Node_AVL_Tree<Type>* temp = new Node_AVL_Tree<Type>;
        temp->data = value;
        temp->height = 1;
        head = temp;
        return;
    }
    // this->count_o++;
    head = head->insert(value, this->count_o);
}

template <class Type>
bool Head_AVL_Tree<Type>::is_contain(Type desired) {
    if (!head)
        return false;
    if (head->data == desired) {
        std::cout << "this element is root" << std::endl;
        return true;
    }
}

```

```

        if (head->left && head->data > desired) {
            std::cout << "find in left " << std::endl;
            return head->left->is_contain(desired, 1);
        }
        if (head->right && head->data < desired) {
            std::cout << "find in right " << std::endl;
            return head->right->is_contain(desired, 1);
        }
        return false;
    }

template <class Type>
void Head_AVL_Tree<Type>::remove(Type to_remove) {
    count_o++;
    head = head->remove(to_remove, this->count_o);
}

#endif

```

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.py

```
import unittest
import subprocess

class Tester(unittest.TestCase):

    def test1(self):
        print("d")
        print("a")
        self.assertIn("Finished right",
subprocess.check_output(["./a.out", "d", "a"],
universal_newlines=True))

    def test2(self):
        print("d")
        print("w")
        self.assertIn("Finished right",
subprocess.check_output(["./a.out", "d", "w"],
universal_newlines=True))

    def test3(self):
        print("i")
        print("a")
        self.assertIn("Finished right",
subprocess.check_output(["./a.out", "i", "a"],
universal_newlines=True))

    def test4(self):
        print("i")
        print("w")
        self.assertIn("Finished right",
subprocess.check_output(["./a.out", "i", "w"],
universal_newlines=True))

if __name__ == '__main__':
    unittest.main()
```

Название файла: average\_value.py

```
def av_val(name):
    av_data = []
    save = name
    for i in range(0, 201, 1):
```



```

        name = name + str(i) + '.txt'
        file = open(name, 'r')
        data = file.read()
        data = list(data.split())

        name = save

        a = 0
        for y in range(1, len(data)):
            a += int(data[y])

        average = a / len(data)
        av_data.append(average)
        file.close()

    return av_data

def print_into_file(data, a):
    name = "data_"

    arr = [1]
    for i in range(50, 10050, 50):
        arr.append(i)
    name = name + str(a) + '.txt'
    f = open(name, 'w')
    for j in range(0, len(data)):
        f.write(str(arr[j]) + " " + str(data[j]) + '\n')
    f.close()

name_d_a = 'result_d_a/result_a_'
name_d_w = 'result_d_w/result_w_'
name_i_a = 'result_i_a/result_a_'
name_i_w = 'result_i_w/result_w_'

av_data_d_a = av_val(name_d_a)
av_data_d_w = av_val(name_d_w)
av_data_i_a = av_val(name_i_a)
av_data_i_w = av_val(name_i_w)

print_into_file(av_data_d_a, 0)
print_into_file(av_data_d_w, 1)
print_into_file(av_data_i_a, 2)
print_into_file(av_data_i_w, 3)

```

Название файла: plot.py

```

import sys
import matplotlib.pyplot as plt
import matplotlib.ticker as ticker

```

```

import math
import numpy as num

file = sys.argv[1]

fs = file + ".txt"

f = open(fs, 'r')

x, y = [0], [0]
li = 0

for li in f:
    row = li.split()
    x.append(float(row[0]))
    y.append(float(row[1]))

f.close()

fig, ax1 = plt.subplots(
    nrows=1, ncols=1,
    figsize=(12, 12)
)

t = num.linspace(0.1, max(x))
a = num.log(t + 1) / num.log(2)

x.pop(0)
y.pop(0)
ax1.plot(x, y, label='Количество вызовов операции')
ax1.plot(t, a, label='log(n)')

ax1.grid(which='major',
        color='k')

ax1.minorticks_on()

ax1.grid(which='minor',
        color='gray',
        linestyle=':')

ax1.legend()

ax1.set_xlabel('Количество элементов в дереве')

if file == "data_0" or file == "data_1":
    ax1.set_ylabel('Количество операций для удаления')
if file == "data_2" or file == "data_3":
    ax1.set_ylabel('Количество операций для вставки')

ax1.set_ylabel('Среднее значение  $O(n)$ ')

fig.set_figwidth(15)

```

```
fig.set_figheight(10)  
plt.show()
```