

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

**КУРСОВАЯ РАБОТА**  
**по дисциплине «Алгоритмы и структуры данных»**  
**Тема: Исследование операций вставки и исключения в AVL-деревьях**

Студент гр. 9381

Судаков Е.В.

Преподаватель

\_\_\_\_\_  
\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

## ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студент Судаков Е.В.

Группа 9381

Тема работы : Исследование операций вставки и исключения в **АВЛ-деревьях**

Исходные данные:

Содержание пояснительной записки:

- титульный лист, лист задания, аннотация, содержание;
- формальная постановка задачи;
- описание алгоритма;
- описание структур данных и функций;
- тестирование;
- исследование;
- программный код (в приложении) с комментариями;
- выводы.

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 31.10.2020

Дата сдачи реферата: 10.12.2020

Дата защиты реферата: 00.12.2020

Студент		Судаков Е.В.
Преподаватель		Фирсов М.А.

## **АННОТАЦИЯ**

В данной курсовой работе производится исследование структуры данных “АВЛ-дерево”, а также операция по вставке и удалению элементов из нее. Исследование проходит с помощью тестов для разных случаев поведения алгоритмов, в среднем, худшем случае. Результатом исследования являются числовые метрики, на основе которых формируется статистика для сравнения с теоретическими оценками.

## **SUMMARY**

This paper is supposed to bring brief investigation on abstract data structure AVL-tree and its insertion and deletion operations. Research provided with tests on different cases of structure behavior, e.g. worst, average. The result is numerical statistics compared with theoretical estimation.

## **СОДЕРЖАНИЕ**

<b>ВВЕДЕНИЕ</b>	<b>5</b>
<b>1. ЗАДАНИЕ</b>	<b>6</b>
<b>2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ</b>	<b>7</b>
2.1. Описание алгоритма	7
2.2. Класс AVL дерева	8
2.3. Описание функций	8
<b>3. Тестирование</b>	<b>11</b>
<b>4. Исследование</b>	<b>15</b>
4.1. Асимптотика работы AVL дерева	16
4.1.1 Вставка	16
4.1.2 Удаление	16
4.2. Планирование исследования	17
4.3. Результаты исследования	18
<b>ЗАКЛЮЧЕНИЕ</b>	<b>20</b>
<b>СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ</b>	<b>20</b>
<b>ПРИЛОЖЕНИЕ А</b>	<b>21</b>

## **ВВЕДЕНИЕ**

Целью данной работы является ознакомление учащегося со структурой данных “АВЛ-дерево”, а также проведения исследования работы алгоритмов на этой структуре данных.

## 1. ЗАДАНИЕ

АВЛ-деревья - вставка и исключение. Исследование (в среднем, в худшем случае)

## 2. ХОД ВЫПОЛНЕНИЯ РАБОТЫ

### 2.1. ОПИСАНИЕ АЛГОРИТМА

Алгоритм добавления элемента следующий:

1. Вставка элемента происходит почти также, как и обычном БДП.  
Спускаемся по дереву вниз, сравнивая элемент для вставки с элементами дерева.
2. После вставки необходимо **сбалансировать** дерево.

Балансировка дерева происходит, когда разница между высотами поддеревьев одного элемента становится равной 2. В таком случае, в зависимости от конфигурации, необходимо провести серию **вращений**.

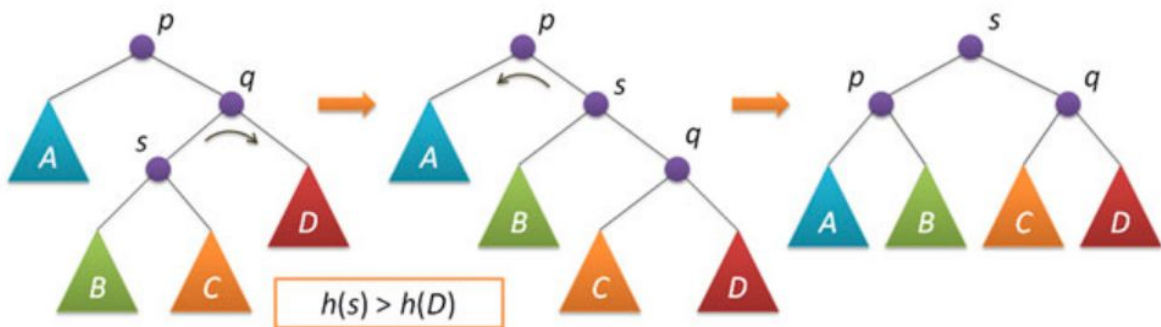


Рисунок 1. Балансировка дерева с помощью правого и левого вращения

Алгоритм удаления элемента :

Находим узел  $p$  с заданным ключом  $k$  (если не находим, то делать ничего не надо), в правом поддереве находим узел  $min$  с наименьшим ключом и заменяем удаляемый узел  $p$  на найденный узел  $min$ . При каждом выходе из рекурсии необходимо ребалансировать дерево.

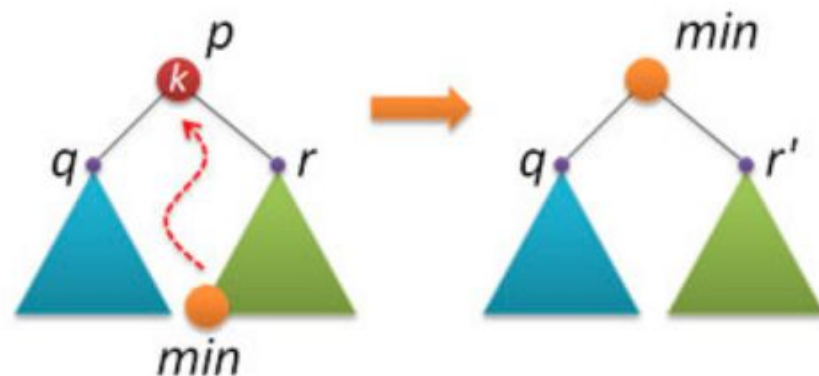


Рисунок 2. Алгоритм удаления элемента

## 2.2. КЛАСС АВЛ ДЕРЕВА

**class AVLTree** - Класс представления АВЛ-дерева. Является по-сути всего лишь синтаксическим сахаром над Node, в то время когда в Node определены функции - вращения, балансировки.

В классе AVLTree определены функции вставки и исключения.

## 2.3. ОПИСАНИЕ ФУНКЦИЙ

Функции описаны в исходном коде в стиле Javadoc:

**int Node::bFactor()** - Функция поиска разности между высотами поддеревьев элемента

@return Разница между высотами левого и правого поддерева

**void Node::updateHeight()** - После каждой вставки/балансировки/удаления нужно обновлять высоту дерева

**Node \*Node::rotateRight()** - Правое вращение.

@return Node\* p - новый корень полученного дерева

**Node \*Node::rotateLeft()** - Функция левого вращения.

@return новый корень дерева

**Node \*Node::balance()** - Функция балансировки АВЛ дерева. Балансировка нужна в случае когда разница высот левого и правого поддеревьев становится  $\geq 2$

@return указатель на самого себя(узла)

**AVLTree::AVLTree(int k)** - Конструктор АВЛ-дерева

@param k ключ для рута

**void AVLTree::printTree(Node \*node, int level)** - Служебная функция вывода дерева. Выводит дерево не сверху-вниз, а слева-направо

@param node корень выводимого поддерева

@param level уровень рекурсии для инdentации



**Node \*AVLTree::insert(Node \*node, int key)** - Вставка элемента.

Единственное отличие от вставки в простое БДПв том, что в конце необходимо балансировать.

@param node корень дерева, куда добавляем

@param key ключ элемента

@return Корень сбалансированного дерева

**Node \*AVLTree::remove(Node \*node, int key)** - Функция удаления элемента с заданным ключом находим узел p с заданным ключом k (если не находим, то делать ничего не надо), в правом поддереве находим узел min с наименьшим ключом и заменяем удаляемый узел p на найденный узел min.

@param node корень дерева, в котором происходит удаление элемента

@param key ключ для удаления

@return ребалансированный корень дерева

**Node \*AVLTree::findMin(Node \*node)** - Функция поиска минимального элемента в (под)дереве

@param node корень дерева, где ищется минимум

@return указатель на элемент с наименьшим ключом

**Node \*AVLTree::removeMin(Node \*node)** - Удаление минимального элемента из заданного дерева. По свойству АВЛ-дерева у минимального элемента справа либо подвешен узел, либо там пусто. В обоих случаях надо просто вернуть указатель на правый узел и по пути назад (при возвращении из рекурсии) выполнить балансировку

@param node корень (под)дерева, где удаляется минимальный элемент

@return указатель на новый корень после балансировки

**Node \*AVLTree::insertNode(Node \*root, int key)** - Служебная функция-обертка над вставком для удобного вывода

@param root корень (под)дерева, куда вставляется элемент

@param key ключ элемента для вставки

@return (возможно обновленный) корень поддерева

**Node \*AVLTree::removeNode(Node \*root, int key)** - Синтаксический сахар над remove.

@param root (под)дерево, в котором удалится элемент

@param key элемент для удаления

@return корень дерева, где удаляли элемент

### 3. ТЕСТИРОВАНИЕ

Основной тест №1:

Входные данные: Создать дерево с корнем 30. Вставить 10. Вставить 11.

Вставить 40. Вставить 35. Вставить 5. Вставить 4. Удалить 11. Вставить 3.

Выходные данные (с промежуточной информацией):

Обратите внимание : дерево выводится слева-направо.

Создано AVL-дерево с корнем 30

30

Введите элемент :

=====

Вставка элемента 10 в дерево с корнем 30

30

10

=====

Введите элемент :

=====

Вставка элемента 11 в дерево с корнем 30

30

11

10

=====

Введите элемент :

=====

Вставка элемента 40 в дерево с корнем 11

40  
30  
11  
10

=====

Введите элемент :

=====

Вставка элемента 35 в дерево с корнем 11

40  
35  
30  
11  
10

=====

Введите элемент :

=====

Вставка элемента 5 в дерево с корнем 11

40  
35  
30  
11

10

5

=====

Введите элемент :

=====

Вставка элемента 4 в дерево с корнем 11

40

35

30

11

10

5

4

=====

Введите элемент :

=====

Удаление элемента 11 из дерева с корнем 11

40

35

30

10

5

4

=====

Введите элемент :

=====

Вставка элемента 3 в дерево с корнем 30

40  
35  
30  
10  
5  
4  
3

=====

Process finished with exit code 0

## 4. ИССЛЕДОВАНИЕ

### 4.1. АСИМПТОТИКА РАБОТЫ АВЛ ДЕРЕВА

#### 4.1.1 ВСТАВКА

Пусть нам надо добавить ключ  $t$ . Будем спускаться по дереву, как при поиске ключа  $t$ . Если мы стоим в вершине  $a$  и нам надо идти в поддереву, которого нет, то делаем ключ  $t$ . листом, а вершину  $a$  его корнем. Дальше поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину  $i$  из левого поддерева, то  $diff[i]$  увеличивается на единицу, если из правого, то уменьшается на единицу. Если пришли в вершину и её баланс стал равным нулю, то это значит высота поддерева не изменилась и подъём останавливается. Если пришли в вершину и её баланс стал равным 1 или  $-1$ , то это значит высота поддерева изменилась и подъём продолжается. Если пришли в вершину и её баланс стал равным 2 или  $-2$ , то делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, то останавливаемся, иначе продолжаем подъём.

Так как в процессе добавления вершины мы рассматриваем не более, чем  $O(h)$  вершин дерева, и для каждой запускаем балансировку не более одного раза, то суммарное количество операций при включении новой вершины в дерево составляет  $O(\log n)$  операций.

#### 4.1.2 УДАЛЕНИЕ

Для простоты опишем рекурсивный алгоритм удаления. Если вершина — лист, то удалим её, иначе найдём самую близкую по значению вершину  $a$ , переместим её на место удаляемой вершины и удалим вершину  $a$ . От удалённой

вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если мы поднялись в вершину  $i$  из левого поддерева, то  $diff[i]$  уменьшается на единицу, если из правого, то увеличивается на единицу. Если пришли в вершину и её баланс стал равным 1 или  $-1$ , то это значит, что высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины стал равным нулю, то высота поддерева уменьшилась и подъём нужно продолжить. Если баланс стал равным 2 или  $-2$ , следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

В результате указанных действий на удаление вершины и балансировку суммарно тратится, как и ранее,  $O(h)$  операций. Таким образом, требуемое количество действий —  $O(\log n)$ .

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

Рисунок 3. Асимптотика работы

## 4.2. План исследования

Для подтверждения теоретической оценки был создан класс Research, который генерирует входные данные двух типов - строго возрастающей последовательности, и случайной. Каждая последовательность подается на вход операции вставки. Далее генерируется набор индексов элементов для удаления, и он подается на вход операции исключения. Во время работы вставки/исключения фиксируется количество вызовов этих функций.



### 4.3. РЕЗУЛЬТАТЫ ИССЛЕДОВАНИЯ

Ниже приведены 4 графика, иллюстрирующие асимптотику выполнения операций в обоих случаях. На всех графиках желтой линией нарисован график логарифма от количества элементов в дереве. Этот график позволяет убедиться, что теоретическая оценка совпадает с практикой.

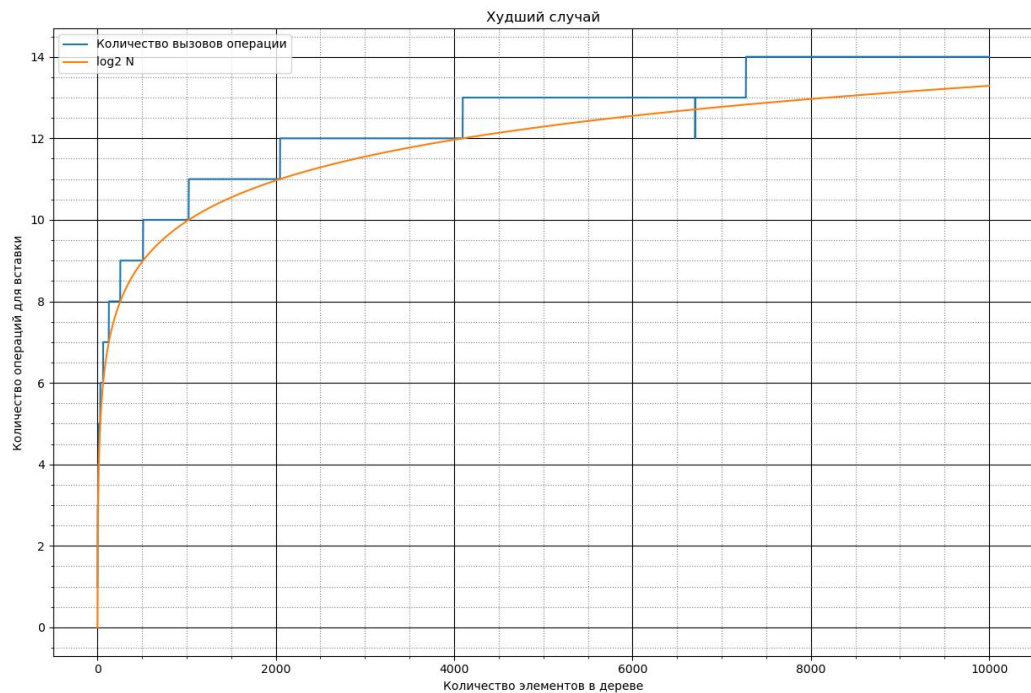


Рисунок 4. Количество вызовов операции вставки в худшем случае.

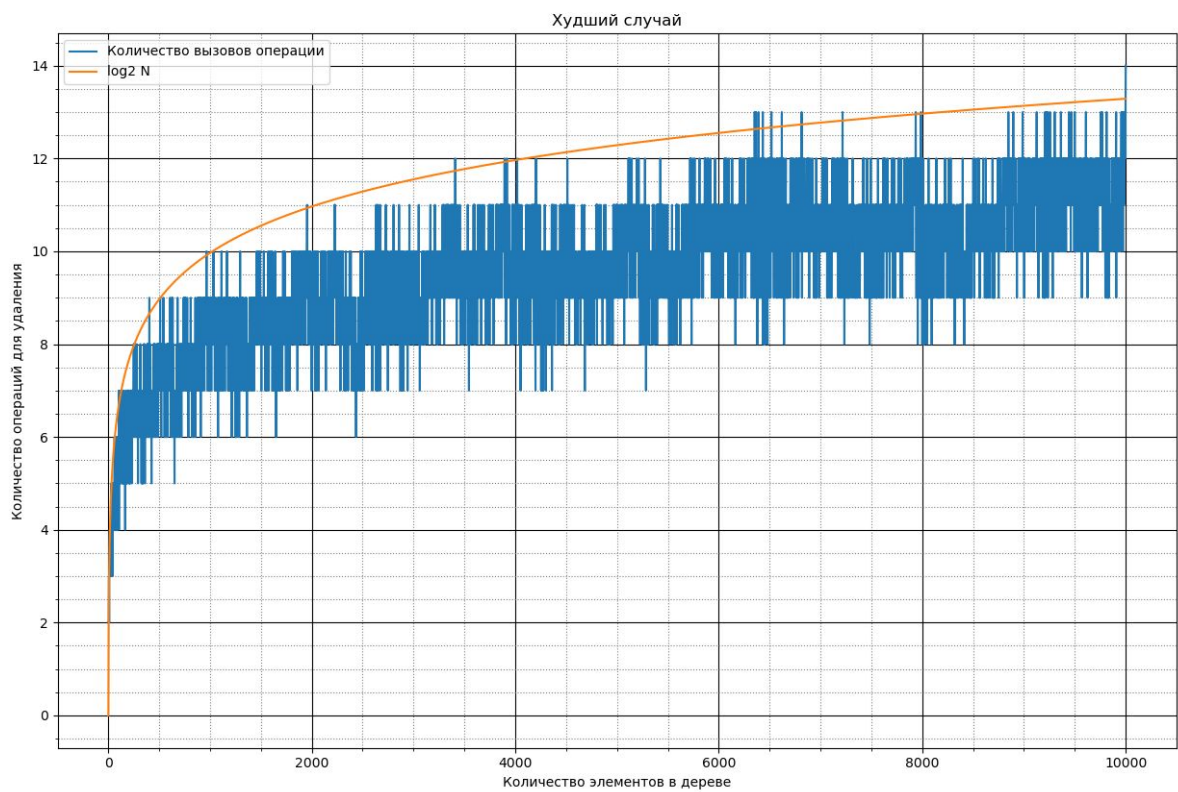


Рисунок 5. Количество вызовов операции удаления в худшем случае.

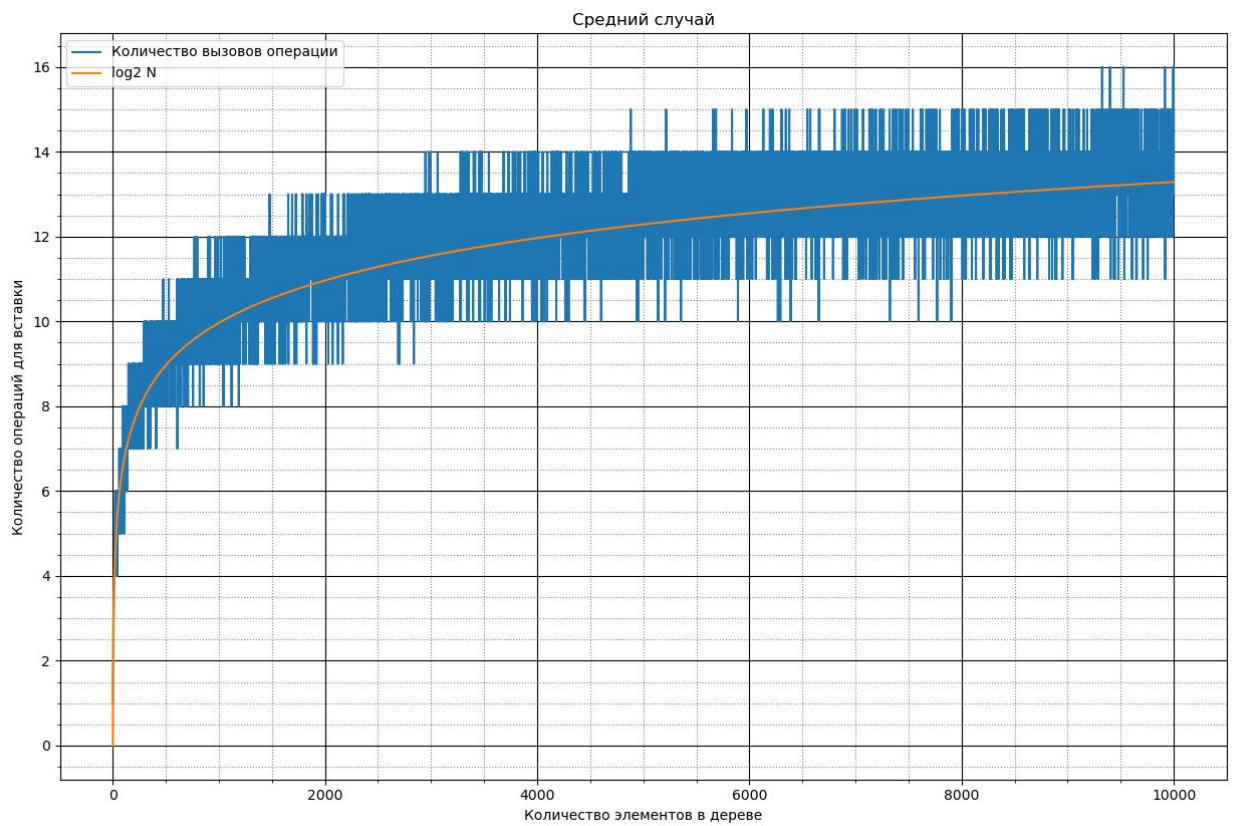
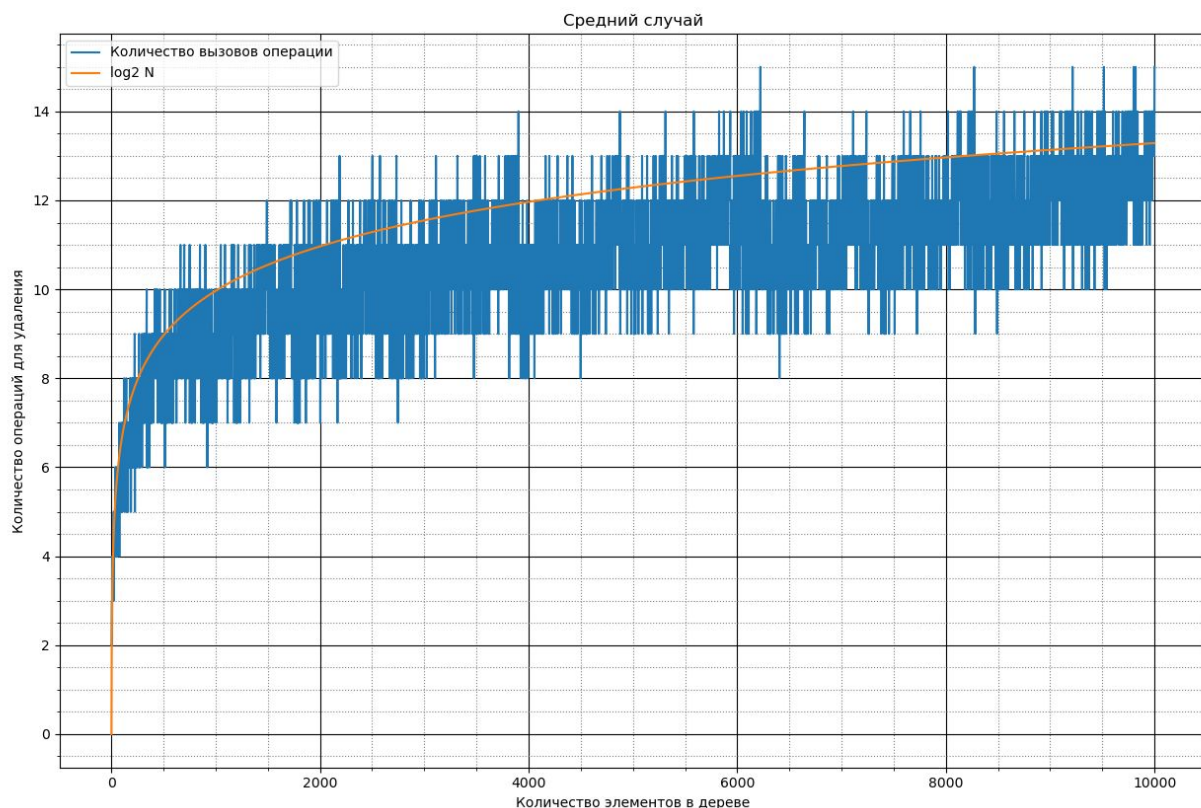


Рисунок 6. Количество вызовов операции вставки в рандомизированном случае.



*Рисунок 6. Количество вызовов операции удаления в рандомизированном случае.*

## ЗАКЛЮЧЕНИЕ

Таким образом, в ходе выполнения курсовой работы наглядным образом была доказана теоретическая оценка асимптотики работы операций вставки и исключения для АВЛ деревьев.

## СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. <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>
2. <https://habr.com/ru/post/150732/>
3. <https://ru.wikipedia.org/wiki/%D0%90%D0%92%D0%9B-%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%BE>

## ПРИЛОЖЕНИЕ А

### Исходный код программы

```
#include <bits/stdc++.h>

using namespace std;

#define inputSize 10000
#define researchAddResultFile "resAdd.txt"
#define researchDeleteResultFile "resDelete.txt"
#define rotationCountFile "rotateFile.txt"
static int operationCount = 0;
static int rotationCount = 0;

#ifdef __linux__
#define REMOVE_COLOR "\033[1m\033[31m"
#define INSERT_COLOR "\033[1m\033[32m"
#define RESET_COLOR "\033[0m"

#elif __WIN32
#define REMOVE_COLOR ""
#define INSERT_COLOR ""
#define RESET_COLOR ""
#endif

/*
 * Вариант 16. БДП: АВЛ-дерево
 * Действия : 1. По заданной последовательности элементов построить
дерево
 * 2 б) : Для построенной структуры данных проверить, входит ли в
неё элемент, и если
 * входит, то удалить элемент из дерева. Предусмотреть возможность
повторного выполнения с другим
 * Элементом
 */

/**
 * Представление узла дерева
 */
class Node {
    int key;
    int height;
    Node *left;
    Node *right;

    Node *rotateRight();

    Node *rotateLeft();

public:
```

```

Node(int k) : key(k), left(0), right(0), height(1) {}

int getHeight();

int bFactor();

void updateHeight();

Node *balance();

Node *getRight();

Node *getLeft();

void setLeft(Node *node);

void setRight(Node *node);

void setKey(int key);

int getKey();

};

int Node::getHeight() {
    return this ? this->height : 0;
}

/**
 *
 * @return Разница между высотами левого и правого поддерев
 */
int Node::bFactor() {
    return this->right->getHeight() - this->left->getHeight();
}

/**
 * После каждой вставки/балансировки/удаления нужно
 * обновлять высоту дерева
 */
void Node::updateHeight() {
    int hl = this->left->getHeight();
    int hr = this->right->getHeight();
    this->height = max(hl, hr) + 1;
}

/**
 * Правое вращение.
 *
 *      y
 *     /\
 *    /\
 *
 *      x
 *     /\
 *    /\
 *
 *      Правое вращение

```

```

*      x      T3      - - - - - >          T1      y
*      / \      < - - - - - - -          / \
*      T1 T2      Левое вращение          T2 T3
* @return Node* p - новый корень полученного дерева
*/
Node *Node::rotateRight() {
    Node *newRoot = this->left;
    this->left = newRoot->right;
    newRoot->right = this;
    this->updateHeight();
    newRoot->updateHeight();
    return newRoot;
}

/**
 * Функция левого вращения.
 * @return новый корень дерева
 */
Node *Node::rotateLeft() {
    Node *newRoot = this->right;
    this->right = newRoot->left;
    newRoot->left = this;
    this->updateHeight();
    newRoot->updateHeight();
    return newRoot;
}

/**
 * Функция балансировки AVL дерева.
 * Балансировка нужна в случае когда разница высот левого и
 * правого поддеревьев становится == |2|
 * @return указатель на самого себя(узла)
 */
Node *Node::balance() {
    rotationCount++;
    this->updateHeight();
    int diff = this->bFactor();
    if (diff == 2) {
        if (this->right->bFactor() < 0) { // высота правого внука
меньше высоты левого внука
            this->right = this->right->rotateRight();
        }
        return this->rotateLeft(); // иначе - правый внук больше
либо равен левому и простое вращение
    } else if (diff == -2) {
        if (this->left->bFactor() > 0) {
            this->left = this->left->rotateLeft();
        }
        return this->rotateRight();
    }
    return this;
}

```

```
Node *Node::getRight() {
    return this ? this->right : nullptr;
}
```

```
Node *Node::getLeft() {
    return this ? this->left : nullptr;
}
```

```
int Node::getKey() {
    return this ? this->key : 0;
}
```

```
void Node::setLeft(Node *node) {
    if (this)
        this->left = node;
}
```

```
void Node::setRight(Node *node) {
    if (this)
        this->right = node;
}
```

```
void Node::setKey(int key) {
    if (this)
        this->key = key;
}
```

```
/**
 * Класс представления AVL-дерева. Является по-сути всего лишь
 * синтаксическим сахаром над Node.
 */
```

```
class AVLTree {
public:
    Node *root;

    AVLTree(int k);

    void printTree(Node *node, int level);

    Node *insert(Node *node, int key);

    Node *findMin(Node *node);

    Node *removeMin(Node *node);

    Node *remove(Node *node, int key);
}
```



```

        Node *insertNode(Node *root, int key);

        Node *removeNode(Node *root, int key);
};

/**
 * Конструктор АВЛ-дерева
 * @param k ключ для рута
 */
AVLTree::AVLTree(int k) {
    cout << INSERT_COLOR << "Создано АВЛ-дерево с корнем " << k <<
"\n\n";
    this->root = new Node(k);
    this->printTree(this->root, 0);
    cout << RESET_COLOR;
}

/**
 * Служебная функция вывода дерева.
 * Выводит дерево не сверху-вниз, а слева-направо
 * @param node корень выводимого поддерева
 * @param level уровень рекурсии для инdentации
 */
void AVLTree::printTree(Node *node, int level) {
    if (node) {
        printTree(node->getRight(), level + 1);
        for (int i = 0; i < level; i++) cout << "    ";
        cout << node->getKey() << endl;
        printTree(node->getLeft(), level + 1);
    }
}

/**
 * Вставка элемента. Единственное отличие от вставки в простое БДП
 * в том, что в конце необходимо балансировать.
 * @param node корень дерева, куда добавляем
 * @param key ключ элемента
 * @return Корень сбалансированного дерева
 */
Node *AVLTree::insert(Node *node, int key) {
    if (node == nullptr) return new Node(key);
    if (key < node->getKey()) {
        node->setLeft(insert(node->getLeft(), key));
    } else if (key > node->getKey()) { // не нужно вставлять
дубликаты, согласно варианту.
        node->setRight(insert(node->getRight(), key));
    }
    operationCount++;
    return node->balance();
}

```

```

/**
 * Функция удаления элемента с заданным ключом
 * находим узел p с заданным ключом k
 * (если не находим, то делать ничего не надо),
 * в правом поддереве находим узел min с наименьшим ключом
 * и заменяем удаляемый узел p на найденный узел min.
 * @param node корень дерева, в котором происходит удаление
элемента
 * @param key ключ для удаления
 * @return ребалансированный корень дерева
 */
Node *AVLTree::remove(Node *node, int key) {
    operationCount++;
    if (node == nullptr) {
        return nullptr;
    }
    if (key < node->getKey()) {
        node->setLeft(removeMin(node->getLeft()));
    } else if (key > node->getKey()) {
        node->setRight(removeMin(node->getRight()));
    } else { // key == node->getKey()
        Node *right = node->getRight();
        Node *left = node->getLeft();
        delete node;
        if (!right) return left;
        Node *min = findMin(right);
        min->setRight(removeMin(right));
        min->setLeft(left);
        return min->balance();
    }
    return node->balance();
}

/**
 * Функция поиска минимального элемента в (под)дереве
 * @param node корень дерева, где ищется минимум
 * @return указатель на элемент с наименьшим ключом
 */
Node *AVLTree::findMin(Node *node) {
    return node->getLeft() ? findMin(node->getLeft()) : node;
}

/**
 * Удаление минимального элемента из заданного дерева.
 * по свойству AVL-дерева у минимального элемента справа
 * либо подвешен узел, либо там пусто.
 * В обоих случаях надо просто вернуть указатель на правый
 * узел и по пути назад (при возвращении из рекурсии)
 * выполнить балансировку
 * @param node корень (под)дерева, где удаляется минимальный
элемент

```

```

    * @return указатель на новый корень после балансировки
    */
Node *AVLTree::removeMin(Node *node) {
    operationCount++;
    if (node->getLeft() == nullptr) {
        return node->getRight();
    }
    node->setLeft(removeMin(node->getLeft()));
    return node->balance();
}

/**
 * Служебная функция-обертка над вставком для удобного вывода
 * @param root корень (под)дерева, куда вставляется элемент
 * @param key ключ элемента для вставки
 * @return (возможно обновленный) корень поддерева
 */
Node *AVLTree::insertNode(Node *root, int key) {
    cout << INSERT_COLOR;
    cout << "\n===== \n";
    cout << "Вставка элемента " << key << " в дерево с корнем " <<
    root->getKey() << "\n\n";
    root = this->insert(root, key);
    this->printTree(root, 0);
    cout << "\n===== \n" << RESET_COLOR;
    return root;
}

/**
 * Синтаксический сахар над remove.
 * @param root (под)дерево, в котором удалится элемент
 * @param key элемент для удаления
 * @return корень дерева, где удаляли элемент
 */
Node *AVLTree::removeNode(Node *root, int key) {
    cout << REMOVE_COLOR;
    cout << "\n===== \n";
    cout << "Удаление элемента " << key << " из дерева с корнем "
    << root->getKey() << "\n\n";
    root = this->remove(root, key);
    this->printTree(root, 0);
    cout << "\n===== \n" << RESET_COLOR;
    return root;
}

void printMenu() {
    cout << "\n===== \n"
        << "(1) Вставить элемент\n"
        << "(2) Удалить элемент\n"
        << "(3) Выход\n\n";
}

```

```

AVLTree *processUserInput(AVLTree *tree) {
    int f, userKey;
    printMenu();
    cin >> f;
    switch (f) {
        case 1:
            cout << "Введите элемент : \n";
            cin >> userKey;
            if (tree) {
                tree->root = tree->insertNode(tree->root,
userKey);
            } else {
                tree = new AVLTree(userKey);
            }
            break;
        case 2:
            if (tree) {
                cout << "Введите элемент : \n";
                cin >> userKey;
                tree->root = tree->removeNode(tree->root,
userKey);
            } else cout << "В дереве нет элементов! \n";
            break;
        case 3:
            exit(0);
    }
    return tree;
}

class Research {
public:
    unordered_set<int> input;

    void generateAscendance();

    void generateRandom(unordered_set<int> &input, int lower, int
upper);

    void runAdd(AVLTree *tree);

    void runDelete(AVLTree *tree);

};

void Research::generateAscendance() {
    for(int i = 1; i <= inputSize; i++) {
        input.insert(i);
    }
}

void Research::generateRandom(unordered_set<int> &input, int
lower, int upper) {
    auto now = std::chrono::high_resolution_clock::now();

```

```

        std::mt19937 gen;
        gen.seed(now.time_since_epoch().count());
        std::uniform_int_distribution<> distribution(lower, upper);
        while(input.size() < inputSize) {
            input.insert(distribution(gen));
        }
    }

void Research::runAdd(AVLTree *tree) {
    int treeSize = 0;
    ofstream outAdd, outRot;
    outAdd.open(researchAddResultFile);
    outRot.open(rotationCountFile);
    for(auto x : this->input) {
        treeSize++;
        operationCount = 0;
        rotationCount = 0;
        tree->root = tree->insert(tree->root, x);
        outAdd << treeSize << ' ' << operationCount << "\n";
        outRot << treeSize << ' ' << rotationCount << "\n";
    }
    outAdd.close();
    outRot.close();
}

void Research::runDelete(AVLTree *tree) {
    ofstream out;
    int treeSize = inputSize;
    out.open(researchDeleteResultFile);
    unordered_set<int> indices;
    this->generateRandom(indices, 1, inputSize);
    for(auto index : indices) {
        operationCount = 0;
        tree->root = tree->remove(tree->root,
tree->root->getKey());
        out << treeSize << ' ' << operationCount << "\n";
        treeSize--;
    }

    out.close();
}

int main() {
    AVLTree *tree = new AVLTree(0);

    // интерактивчик
    // while (true) {
    //     tree = processUserInput(tree);
    // }
}

```

```
Research res;  
//res.generateRandom(res.input, 0, inputSize);  
res.generateAscendance();  
res.runAdd(tree);  
res.runDelete(tree);  
  
return 0;  
}
```