

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

Студентка гр.9382

Балаева М.О.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

ЗАДАНИЕ НА КУРСОВУЮ РАБОТУ

Студентка Балаева М.О.

Группа 9382

Тема работы : Исследование операций вставки и исключения в AVL-деревьях

Исходные данные:

"Демонстрация" - визуализация структур данных, алгоритмов, действий.
Демонстрация должна быть подробной и понятной (в том числе сопровождаться пояснениями), чтобы программу можно было использовать в обучении для объяснения используемой структуры данных и выполняемых с ней действий

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);
- выводы.

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 09.09.2020

Дата сдачи реферата: 28.12.2020

Дата защиты реферата: 29.12.2020

Студентка		Балаева М.О.
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

In this course work, the data structure “AVL-tree ” is studied, as well as the operation of inserting and removing elements from it. The study is conducted using tests for different cases of algorithm behavior, in the average, worst case. The result of the study is numerical metrics, on the basis of which statistics are formed for comparison with theoretical estimates

ВВЕДЕНИЕ

Двоичное дерево поиска — это двоичное дерево, для которого выполняются следующие дополнительные условия (*свойства дерева поиска*):

- Оба поддерева — левое и правое — являются двоичными деревьями поиска.
- У всех узлов левого поддерева произвольного узла X значения ключей данных меньше либо равны, нежели значение ключа данных самого узла X .
- У всех узлов правого поддерева произвольного узла X значения ключей данных больше либо равны, нежели значение ключа данных самого узла X .

Очевидно, данные в каждом узле должны обладать ключами, на которых определена операция сравнения меньше.

АВЛ-дерево — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1. АВЛ — аббревиатура, образованная первыми буквами фамилий создателей (советских учёных) Георгия Максимовича Адельсон-Вельского и Евгения Михайловича Ландиса.

1. ЗАДАНИЕ

Вариант 16. Реализовать структуру данных “АВЛ-дерево” и провести исследование работы операций вставки и исключения(в среднем, в худшем случае) дабы подтвердить теоретическую оценку работы этих операций.

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.

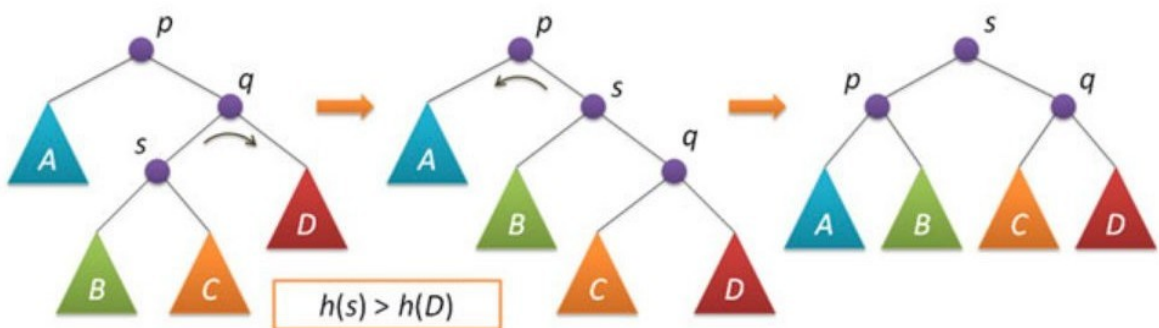


Рисунок 1. Балансировка дерева с помощью правого и левого вращения

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.

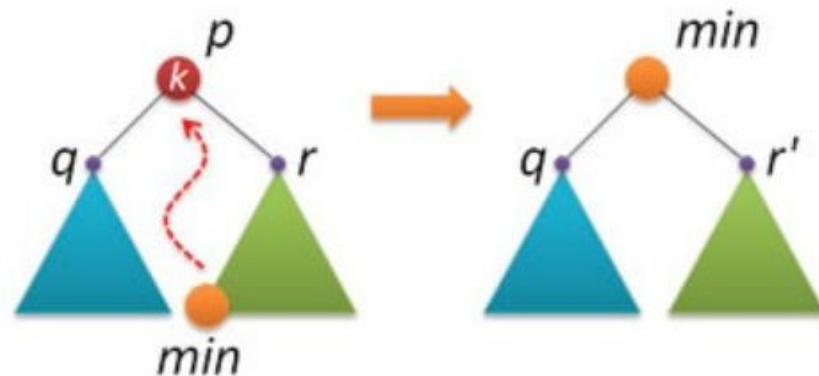


Рисунок 2. Алгоритм удаления элемента

2.2. КЛАСС АВЛ-ДЕРЕВА

class AVL_tree -Класс представления АВЛ-дерева. Является классом-оберткой над Node , в то время когда в Node определены функции - вращения, балансировки.

В классе AVL_tree определены функции вставки и исключения.

2.3. ОПИСАНИЕ ФУНКЦИЙ

Функции описаны ниже.

int diff_height - Функция поиска разности между высотами поддеревьев элемента.

Возвращает разность высот левого и правого поддерева

void update_height - После каждой вставки/балансировки/удаления нужно обновлять высоту дерева

Node *rotate_right - Правое вращение.

Возвращает Node* p - новый корень полученного дерева

Node *rotate_left - Функция левого вращения.

Возвращает новый корень дерева

Node *balance - Функция балансировки АВЛ-дерева. Балансировка нужна в случае когда разница высот левого и правого поддеревьев становится равной |2|.

Возвращает указатель на самого себя(узел).

AVL_tree - Конструктор АВЛ-дерева принимает корень.

void print_tree - Служебная функция вывода дерева. Выводит дерево не сверху-вниз, а слева-направо.

Принимает корень выводимого поддерева.

Принимает уровень рекурсии для индентации.

Node *insert_node - Вставка элемента. В конце необходимо балансировать.

Принимает корень дерева, куда добавляем.

Принимает ключ элемента.

Возвращает корень сбалансированного дерева.

Node *remove_node - Функция удаления элемента с заданным ключом находим узел *p* с заданным ключом *value* , в правом поддереве находим узел *min* с наименьшим ключом и заменяем удаляемый узел *p* на найденный узел *min*.

Принимает корень дерева, в котором происходит удаление элемента.

Принимает *value* ключ для удаления.

Возвращает ребалансированный корень дерева.

Node *find_min - Функция поиска минимального элемента в дереве или поддереве.

Возвращает корень дерева, где ищется минимум.

Возвращает указатель на элемент с наименьшим ключем.

Node *remove_min - 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

при возвращении из рекурсии выполнить

балансировку.

Принимает корень дерева или поддереву, где

удаляется минимальный элемент.

Возвращает указатель на новый корень после

балансировки.

Node *lets_insert_node - Служебная функция-обертка над вставком для удобного вывода.

Возвращает корень дерева или поддереву, куда

вставляется элемент.

Принимает ключ элемента для вставки.

Возвращает корень поддереву.

Node *lets_remove_node- Служебная функция-обертка над remove.

Принимает поддерево или дерево, в котором удаляется

элемент ,элемент для удаления. Возвращает корень

дерева, где удаляли элемент.

3. ТЕСТИРОВАНИЕ

Основной тест №1:

Входные данные: Создать дерево с корнем 35. Вставить 11. Вставить 10.

Вставить 40. Вставить 30. Вставить 6. Вставить 4. Удалить 10. Вставить 3.

Выходные данные (с промежуточной информацией):

Обратите внимание : дерево выводится слева-направо.

Создано AVL-дерево с корнем.

Таблица.

№	Входные данные	Выходные данные
1.	1 35	-!-!-!-!-!-!-!-!-!-! 35 -!-!-!-!-!-!-!-!-!-
2.	1 11	-!-!-!-!-!-!-!-!-!-! 35 11 -!-!-!-!-!-!-!-!-!-
3.	1 10	-!-!-!-!-!-!-!-!-!-! 35 11 10 -!-!-!-!-!-!-!-!-!-
4.	1 40	-!-!-!-!-!-!-!-!-!-! 40 35 11 10 -!-!-!-!-!-!-!-!-!-
5.	1 30	-!-!-!-!-!-!-!-!-!-! 40 35 30 11 10 -!-!-!-!-!-!-!-!-!-
6.	1 6	-!-!-!-!-!-!-!-!-!-! 40 35 30

		11 10 6 -!-!-!-!-!-!-!-!-!-
7.	1 4	-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 6 4 -!-!-!-!-!-!-!-!-!
8.	2 10	-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 6 -!-!-!-!-!-!-!-!-!
9.	1 3	-!-!-!-!-!-!-!-!-!- 40 35 30 11 10 6 3 -!-!-!-!-!-!-!-!-!

4. ИССЛЕДОВАНИЕ

4.1. АСИМПТОТИКА РАБОТЫ АВЛ-ДЕРЕВА.

4.1.1 ВСТАВКА.

Добавим ключ t . Будем спускаться по дереву, как при поиске ключа. Если находимся в вершине a , следовательно нужно идти в поддереву, если его нет, делаем t листом, а вершину a его корнем. Далее поднимаемся вверх по пути поиска и пересчитываем баланс у вершин. Если мы поднялись в вершину из левого поддерева, увеличиваем на единицу, если из правого, уменьшаем на единицу. Если пришли в вершину, её баланс стал равен нулю, значит высота поддерева не изменилась и подъём останавливается, если пришли в вершину и её баланс стал равным 1 или -1 , значит высота поддерева изменилась и подъём продолжается. Если пришли в вершину, её баланс стал равным 2 или -2 , делаем одно из четырёх вращений и, если после вращения баланс стал равным нулю, останавливаемся, иначе продолжаем подъём.

Можно сделать вывод, что при добавлении 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.

Если вершина — лист, то удалим её, если нет, найдём самую близкую по значению вершину, переместим её на место удаляемой вершины и удалим ее. От удалённой вершины будем подниматься вверх к корню и пересчитывать баланс у вершин. Если поднялись в вершину из левого поддерева, то уменьшаем значение вершины на единицу, если из правого, то увеличивается на единицу. Если

баланс вершины равен 1 или -1 , значит, высота этого поддерева не изменилась и подъём можно остановить. Если баланс вершины равен нулю, высота поддерева уменьшилась и подъём нужно продолжить. Если баланс баланс 2 или -2 , следует выполнить одно из четырёх вращений и, если после вращений баланс вершины стал равным нулю, то подъём продолжается, иначе останавливается.

Получаем, что на удаление вершины и балансировку суммарно тратится, $O(h)$ операций. Таким образом, требуемое количество действий — $O(\log n)$.

4.2. ПЛАН ИССЛЕДОВАНИЯ.

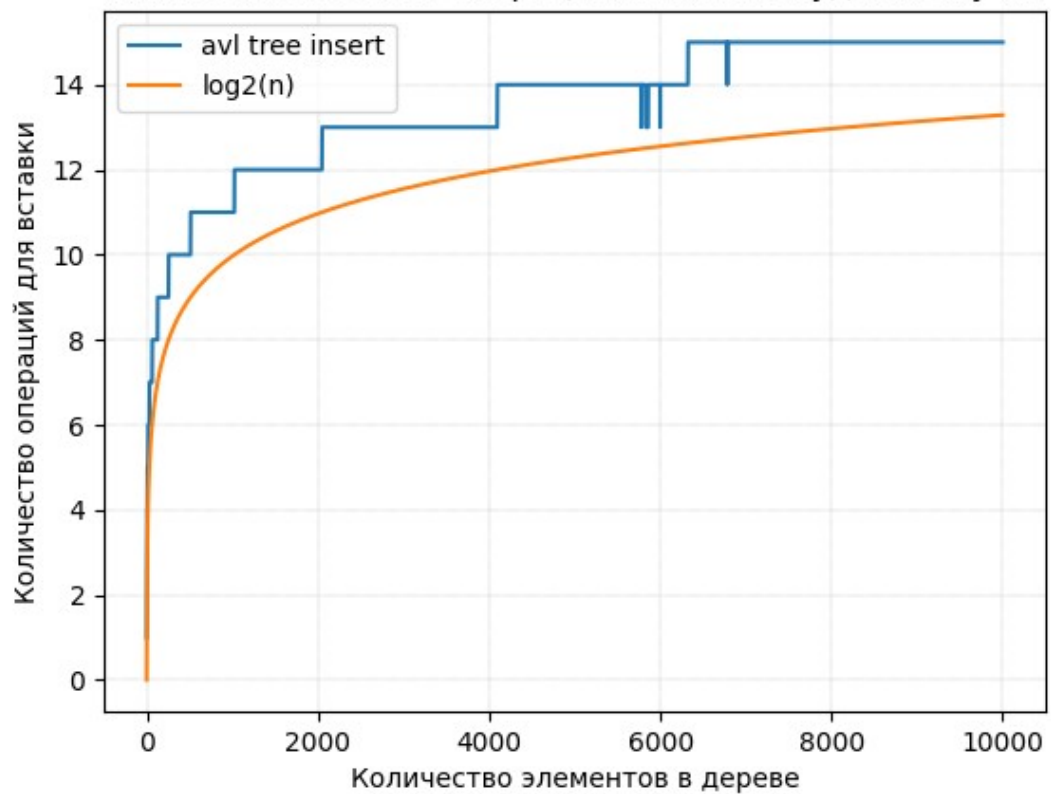
Был создан класс `Research`, который генерирует входные данные двух типов - строго возрастающей последовательности, и случайной. Каждая последовательность подается на вход операции вставки, потом генерируется набор индексов элементов для удаления. Этот набор подается на вход операции исключения. Во время работы фиксируется количество вызовов функций вставки, удаления.

4.3. РЕЗУЛЬТАТЫ ИССЛЕДОВАНИЯ.

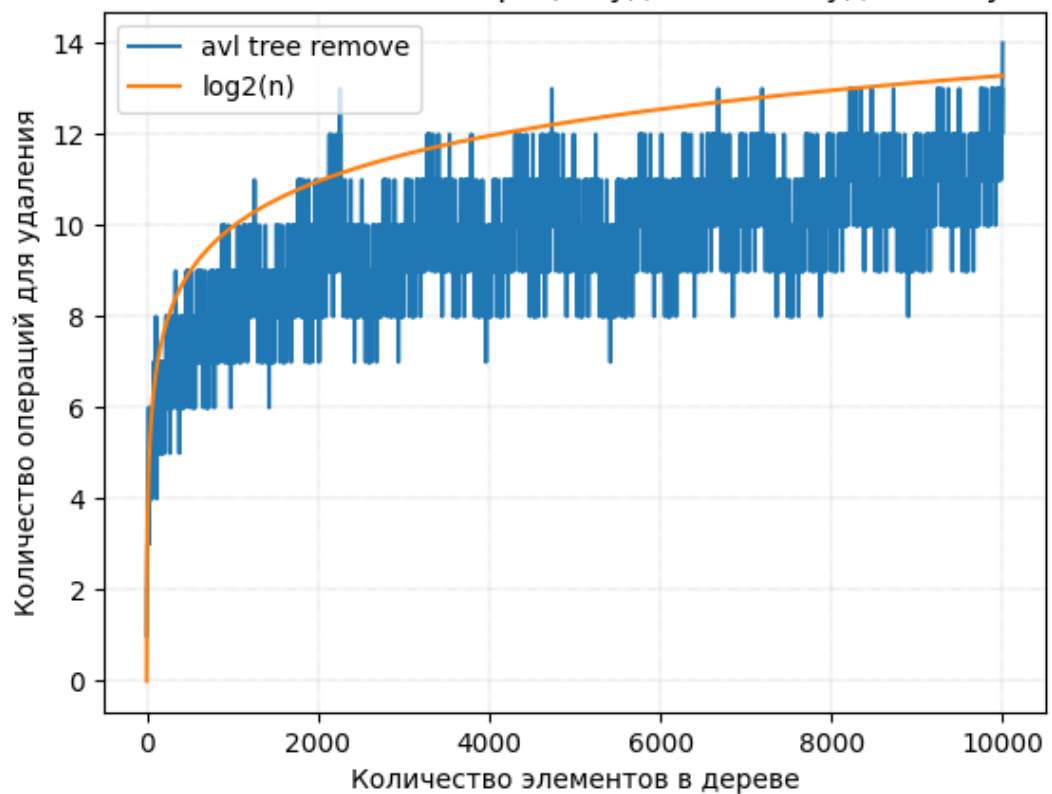
Ниже приведены графики, иллюстрирующие асимптотику выполнения операций в обоих случаях. На всех графиках оранжевой линией нарисован график логарифма от количества элементов в дереве. Этот график позволяет убедиться, что теоретическая оценка совпадает с практикой.

Визуализация была выполнена с помощью графиков, построенных с помощью `python3`.

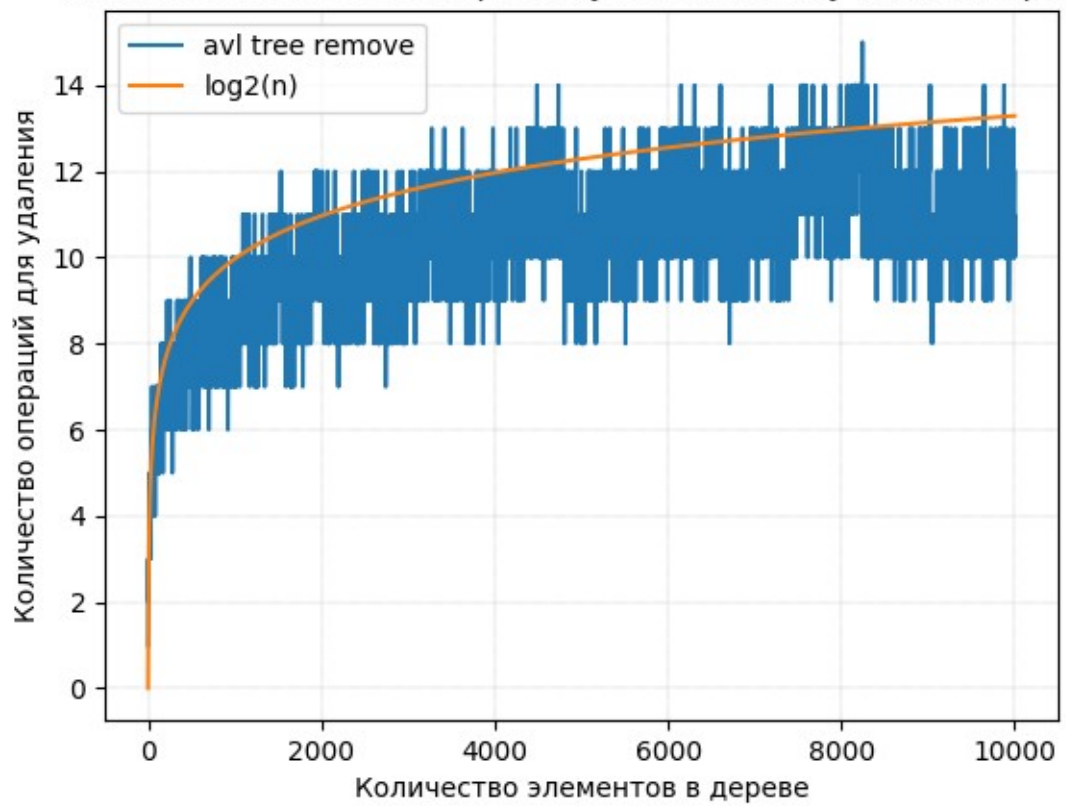
Количество вызовов операции вставки в худшем случае

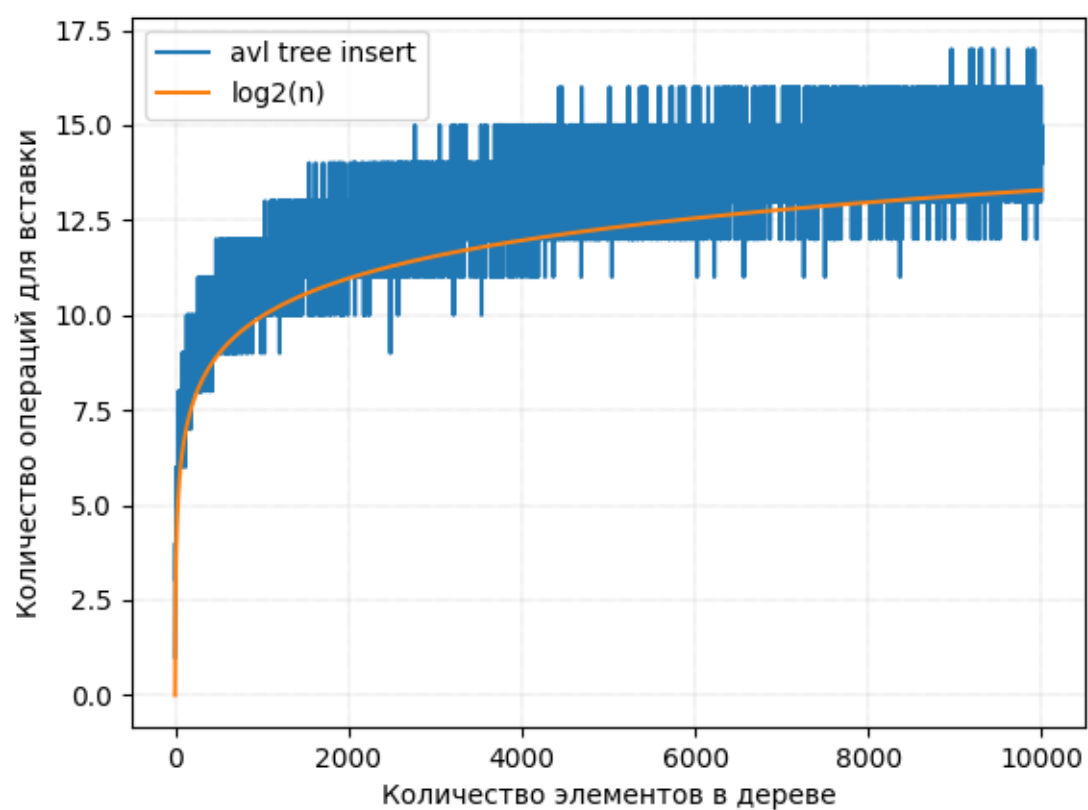


Количество вызовов операции удаления в худшем случае



Количество вызовов операции удаления в случайном дереве





Заключение

В ходе выполнения курсовой работы была доказана теоретическая оценка асимптотики работы операций вставки и исключения для AVL деревьев.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. <https://monster-book.com/grokaem-algoritmy>
2. <https://habr.com/ru/post/150732/>
3. <https://monster-book.com/algoritmy-teoriya-i-prakticheskoe-primenenie>

Приложение

main.cpp

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

using namespace std;
static int op_count = 0;
static int rot_count = 0;
class Node {
protected:
    int value;
    int height;
    Node *lt;
    Node *rt;
    Node *rotate_right();
    Node *rotate_left();

public:
    Node(int k) : value(k), lt(nullptr), rt(nullptr),
height(1) {}
    int get_height();
    int diff_height();
    void update_height();
    Node *balance();
    Node *get_right();
    Node *get_left();
    void set_left(Node *node);
    void set_right(Node *node);
    void set_value(int value);
    int get_value();
};

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

int Node::diff_height() {
    return this->rt->get_height() - this->lt->get_height();
}

void Node::update_height() {
    int hl = this->lt->get_height();
    int hr = this->rt->get_height();
    this->height = max(hl, hr) + 1;
}

Node *Node::rotate_right() {
    Node *new_root = this->lt;
    this->lt = new_root->rt;
    new_root->rt = this;
    this->update_height();
    new_root->update_height();
}
```

```

        return new_root;
    }

Node *Node::rotate_left() {
    Node *new_root = this->rt;
    this->rt = new_root->lt;
    new_root->lt = this;
    this->update_height();
    new_root->update_height();
    return new_root;
}

Node *Node::balance() {
    rot_count++;
    this->update_height();
    int diff = this->diff_height();
    if (diff == 2) {
        if (this->rt->diff_height() < 0) this->rt = this->rt-
>rotate_right();
        return this->rotate_left();
    } else if (diff == -2) {
        if (this->lt->diff_height() > 0) this->lt = this->lt-
>rotate_left();
        return this->rotate_right();
    }
    return this;
}

Node *Node::get_right() {
    return this ? this->rt : nullptr;
}

Node *Node::get_left() {
    return this ? this->lt : nullptr;
}

int Node::get_value() {
    return this ? this->value : 0;
}

void Node::set_left(Node *node) {
    if (this)
        this->lt = node;
}

void Node::set_right(Node *node) {
    if (this)
        this->rt = node;
}

void Node::set_value(int value) {
    if (this)
        this->value = value;
}

```

```

}

class AVL_tree {
public:
    Node *root;
    AVL_tree() : root(nullptr) {};
    AVL_tree(int k);
    void print_tree(Node *node, int level);
    Node *insert_node(Node *node, int value);
    Node *find_min(Node *node);
    Node *remove_min(Node *node);
    Node *remove_node(Node *node, int value);
    Node *lets_insert_node(Node *root, int value);
    Node *lets_remove_node(Node *root, int value);
};

AVL_tree::AVL_tree(int k) {
    cout << "[Created avl tree| root:" << k << "]\n\n";
    this->root = new Node(k);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!" << endl;
    this->print_tree(this->root, 0);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!" << endl;
}

void AVL_tree::print_tree(Node *node, int level) {
    if (node) {
        print_tree(node->get_right(), level + 1);
        for (int i = 0; i < level; i++) cout << "    ";
        cout << node->get_value() << endl;
        print_tree(node->get_left(), level + 1);
    }
}

Node *AVL_tree::insert_node(Node *node, int value) {
    op_count++;
    if (node == nullptr) return new Node(value);
    if (value < node->get_value()) {
        node->set_left(insert_node(node->get_left(), value));
    } else if (value > node->get_value()) {
        node->set_right(insert_node(node->get_right(),
value));
    }
    return node->balance();
}

Node *AVL_tree::remove_node(Node *node, int value) {
    op_count++;
    if (node == nullptr) {
        return nullptr;
    }
    if (value < node->get_value()) {
        node->set_left(remove_min(node->get_left()));
    } else if (value > node->get_value()) {

```

```

        node->set_right(remove_min(node->get_right()));
    } else {
        Node *rt = node->get_right();
        Node *lt = node->get_left();
        delete node;
        if (!rt) return lt;
        Node *min = find_min(rt);
        min->set_right(remove_min(rt));
        min->set_left(lt);
        return min->balance();
    }
    return node->balance();
}

Node *AVL_tree::find_min(Node *node) {
    return node->get_left() ? find_min(node->get_left()) :
node;
}

Node *AVL_tree::remove_min(Node *node) {
    op_count++;
    if (node->get_left() == nullptr) {
        return node->get_right();
    }
    node->set_left(remove_min(node->get_left()));
    return node->balance();
}

Node *AVL_tree::lets_insert_node(Node *root, int value) {
    cout << "[Insert element:" << value << "]\n\n";
    root = this->insert_node(root, value);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!" << endl;
    this->print_tree(root, 0);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!" << endl;
    return root;
}

Node *AVL_tree::lets_remove_node(Node *root, int value) {
    cout << "[Remove element:" << value << "]\n\n";
    root = this->remove_node(root, value);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!" << endl;
    this->print_tree(root, 0);
    cout << "-!-!-!-!-!-!-!-!-!-!-!-!" << endl;
    return root;
}

void print_menu() {
    cout << "1.Insert element\n"
           "2.Remove element\n"
           "3.Exit\n\n";
}

AVL_tree *process_user_input(AVL_tree *tree) {

```

```

    int f, user_value;
    print_menu();
    cin >> f;
    switch (f) {
        case 1:
            cout << "Enter element: \n";
            cin >> user_value;
            if (tree) {
                tree->root = tree->lets_insert_node(tree-
>root, user_value);
            } else {
                tree = new AVL_tree(user_value);
            }
            break;
        case 2:
            if (tree) {
                cout << "Enter element: \n";
                cin >> user_value;
                tree->root = tree->lets_remove_node(tree-
>root, user_value);
            } else cout << "Tree is empty! \n";
            break;
        case 3:
            exit(0);
    }
    return tree;
}

class Research {
    int input_size;
public:
    unordered_set<int> input;
    Research(int v = 10000) : input_size(v) {};
    void generate_ascendance();

    void generate_random(int lower, int upper);

    void run_add(AVL_tree *tree);

    void run_delete(AVL_tree *tree);

};

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

void Research::generate_random(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() < input_size) {
            input.insert(distribution(gen));
        }
    }

void Research::run_add(AVL_tree *tree) {
    int tree_size = 0;
    ofstream out;
    out.open("../research_add.csv");
    out << "tree_size," << "op_count," << "rot_count" << endl;
    for(auto x : this->input) {
        op_count = 0;
        rot_count = 0;
        tree_size++;
        tree->root = tree->insert_node(tree->root, x);
        out << tree_size << ',' << op_count << "," <<
rot_count << "\n";
    }
    out.close();
}

void Research::run_delete(AVL_tree *tree) {
    ofstream out;
    int tree_size = input_size;
    out.open("../research_delete.csv");
    out << "tree_size," << "op_count," << "rot_count" << endl;
    for(auto index : input) {
        op_count = 0;
        rot_count = 0;
        tree->root = tree->remove_node(tree->root, tree->root-
>get_value());
        out << tree_size << ',' << op_count << "," <<
rot_count << "\n";
        tree_size--;
    }
    out.close();
}

int main() {
    AVL_tree *tree = new AVL_tree();
    /*while (true) {
        tree = process_user_input(tree);
    }*/
    Research res;
    res.generate_random(0, 10000);
    // res.generate_ascendance();
    res.run_add(tree);
    res.run_delete(tree);
    Return 0;
}

```



```
}
```

grafiki.py

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

lg = {
    'count': [x for x in range(1, 10000, 1)],
    'val': [np.log2(x) for x in range(1, 10000, 1)]
}

df1 = pd.read_csv('research_add.csv')
df2 = pd.read_csv('research_delete.csv')

fig = plt.figure()
fig.subplots_adjust()

ax1 = fig.add_subplot(111)
# plt.title('Количество вызовов операции вставки в худшем случае')
# plt.title('Количество вызовов операции вставки в случайном дереве')
ax1.set_xlabel('Количество элементов в дереве')
ax1.set_ylabel('Количество операций для вставки')
ax1.plot(df1['tree_size'], df1['op_count'])
ax1.plot(lg['count'], lg['val'])
ax1.grid(color='gray', linestyle=':', linewidth=0.3)
ax1.legend(['avl tree insert', 'log2(n)'], loc=2)
# plt.savefig("research_add.png")
plt.savefig("research_add_random.png")

# ax1 = fig.add_subplot(111)
# plt.title('Количество вызовов операции удаления в худшем случае')
# plt.title('Количество вызовов операции удаления в случайном дереве')
# ax1.set_xlabel('Количество элементов в дереве')
# ax1.set_ylabel('Количество операций для удаления')
# ax1.plot(df2['tree_size'], df2['op_count'])
# ax1.plot(lg['count'], lg['val'])
# ax1.grid(color='gray', linestyle=':', linewidth=0.3)
# ax1.legend(['avl tree remove', 'log2(n)'], loc=2)
# plt.savefig("research_delete.png")
# plt.savefig("research_delete_random.png")
plt.show()
```