Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Российский химико-технологический университет имени Д.И. Менделеева»
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7

Вариант 1

Выполнила студентка группы .....КС-33..... Георгиевская Анастасия Игоревна
Ссылка на репозиторий: .....
https://github.com/MUCTR-IKT-CPP/GeorgievskayaAA_33_alg/tree/main/lab%207

Приняли: .....Пысин Максим Дмитриевич
.....Лобанов Алексей Владимирович

Дата сдачи: 26.03.2025

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить:

Декартово дерево (https://habr.com/ru/post/101818/)

Для этого его потребуется реализовать и сравнить в работе с реализованным ранее AVL-деревом

Для анализа работы алгоритма понадобиться провести серии тестов:

В одной серии тестов проводится 50 повторений

Требуется провести серии тестов для N = 2^i элементов, при этом i от 10 до 18 включительно.

В рамках одной серии понадобится сделать следующее:

- Генерируем N случайных значений.
- Заполнить два дерева N количеством элементов в одинаковом порядке.
- Для каждого из серий тестов замерить максимальную глубину полученного деревьев.
- Для каждого дерева после заполнения провести 1000 операций вставки и замерить время.
- Для каждого дерева после заполнения провести 1000 операций удаления и замерить время.
- Для каждого дерева после заполнения провести 1000 операций поиска.
- Для каждого дерева замерить глубины всех веток дерева.

Для анализа структуры потребуется построить следующие графики:

* График зависимости среднего времени вставки от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График зависимости среднего времени удаления от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График зависимости среднего времени поиска от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.
* График максимальной высоты полученного дерева в зависимости от N.
* Гистограмму среднего распределения максимальной высоты для последней серии тестов для AVL и для вашего варианта.
* Гистограмму среднего распределения высот веток в AVL дереве и для вашего варианта, для последней серии тестов.

Задания со звездочкой = + 5 дополнительных первичных баллов:

* Аналогичная серия тестов и сравнение ее для отсортированного заранее набора данных
* Реализовать красно черное дерево и провести все те же проверки с ним.

Описание структуры «Декартово дерево».

Декартово дерево — это структура данных, которая комбинирует особенности бинарного дерева поиска и кучи.

Основные характеристики декартова дерева:

1. Бинарное дерево поиска (BST): элементы дерева упорядочены по ключу. Это значит, что для каждого узла его левый потомок имеет меньший ключ, а правый — больший ключ.
2. Куча: дополнительно, декартово дерево соблюдает свойство кучи, то есть каждый узел дерева имеет приоритет, и приоритет родителя не меньше приоритета его детей.

Это позволяет эффективно выполнять операции вставки, удаления и поиска с гарантированным временем работы O(log(n)).

Структура узла декартова дерева:

Каждый узел в декартовом дереве обычно имеет следующие данные:

- Ключ — значение, по которому производится упорядочивание в дереве (для поиска, вставки и удаления).
- Приоритет — случайное число, которое помогает соблюдать свойство кучи.
- Левый и правый потомки — указатели на дочерние узлы.

Преимущества:

- Балансировка: Декартово дерево гарантирует логарифмическое время работы для основных операций, даже если последовательность вставок и удалений не сбалансирована.
- Простота реализации: благодаря своей структуре и использованию случайных приоритетов, оно не требует явного соблюдения баланса, как, например, AVL- дерево.

Выполнение задачи.

Для выполнения задачи лабораторной был использован язык C++. Реализация AVL-дерева была взята из предыдущей лабораторной работы. Поэтому здесь будет приведена реализация декартова дерева и основной функции

```
    class Treap {
    public:
        struct Node {
            int key, priority;
            Node* left;
            Node* right;

            Node(int key) : key(key), priority(rand()), left(nullptr),
right(nullptr) {}
        };

        Treap() : root(nullptr), size(0) {}

        void insert(int key) {
            root = insert(root, key);
            ++size;
        }

        void remove(int key) {
            root = remove(root, key);
            --size;
        }

        bool search(int key) {
            return search(root, key);
        }

        int getMaxDepth() {
            return getMaxDepth(root);
        }

        void printDepths() {
            printDepths(root, 1);
        }

        int getSize() {
            return size;
        }

        std::vector<int> getAllDepths() {
            std::vector<int> depths;
            collectDepths(root, 1, depths);
            return depths;
```

```
    }

    template <typename Func>
    double measureTime(Func func, int operations) {
        auto start = std::chrono::high_resolution_clock::now();
        for (int i = 0; i < operations; ++i) {
            func(rand() % 10000);
        }
        auto end = std::chrono::high_resolution_clock::now();
        return   std::chrono::duration<double>(end   -   start).count()   /
operations;
    }

  private:
    Node* root;
    int size;

    Node* insert(Node* node, int key) {
        if (!node) return new Node(key);

        if (key < node->key) node->left = insert(node->left, key);
        else node->right = insert(node->right, key);

        if (node->left && node->left->priority > node->priority) {
            node = rotateRight(node);
        }
        else if (node->right && node->right->priority > node->priority) {
            node = rotateLeft(node);
        }

        return node;
    }

    Node* remove(Node* node, int key) {
        if (!node) return node;

        if (key < node->key) node->left = remove(node->left, key);
        else if (key > node->key) node->right = remove(node->right, key);
        else {
            if (!node->left || !node->right) {
                Node* temp = node->left ? node->left : node->right;
                delete node;
                return temp;
            }
            if (node->left->priority > node->right->priority) {
                node = rotateRight(node);
                node->right = remove(node->right, key);
            }
            else {
                node = rotateLeft(node);
                node->left = remove(node->left, key);
            }
```

```
        }
        return node;
    }

    bool search(Node* node, int key) {
        if (!node) return false;
        if (key == node->key) return true;
        if (key < node->key) return search(node->left, key);
        return search(node->right, key);
    }

    Node* rotateLeft(Node* node) {
        Node* newRoot = node->right;
        node->right = newRoot->left;
        newRoot->left = node;
        return newRoot;
    }

    Node* rotateRight(Node* node) {
        Node* newRoot = node->left;
        node->left = newRoot->right;
        newRoot->right = node;
        return newRoot;
    }

    int getMaxDepth(Node* node) {
        if (!node) return 0;
        return 1 + std::max(getMaxDepth(node->left), getMaxDepth(node->right));
    }

    void printDepths(Node* node, int depth) {
        if (!node) return;
        std::cout << "Node " << node->key << " depth: " << depth << std::endl;
        printDepths(node->left, depth + 1);
        printDepths(node->right, depth + 1);
    }

    void collectDepths(Node* node, int currentDepth, std::vector<int>& depths) {
        if (!node) return;
        depths.push_back(currentDepth);
        collectDepths(node->left, currentDepth + 1, depths);
        collectDepths(node->right, currentDepth + 1, depths);
    }
};
```

- Класс Treap. Реализация декартова дерева
    - Вход: Функции конструктора и методы класса
        - Treap() — создаёт пустое дерево (треп) с корнем nullptr и размером 0.
        - Node(int key) — создаёт узел с ключом и случайным приоритетом.

- Работа:
  - insert(int key) — вставляет элемент в дерево, с учётом приоритета узлов выполняются вращения для балансировки.
  - remove(int key) — удаляет элемент из дерева, с учётом приоритета узлов выполняются вращения для балансировки.
  - search(int key) — выполняет поиск элемента в дереве.
  - getMaxDepth() — возвращает максимальную глубину дерева.
  - printDepths() — выводит глубину каждого узла дерева.
  - getSize() — возвращает количество элементов в дереве.
  - getAllDepths() — собирает и возвращает вектор всех глубин узлов дерева.
  - measureTime(Func func, int operations) — измеряет среднее время выполнения операции вставки, удаления или поиска.
- Выход:
  - insert() и remove() изменяют структуру дерева, обновляют его корень и размер.
  - search() возвращает true или false в зависимости от результата поиска.
  - getMaxDepth() возвращает максимальную глубину дерева.
  - printDepths() выводит данные о глубине каждого узла.
  - getSize() возвращает размер дерева.
  - getAllDepths() возвращает вектор с глубинами всех узлов.
  - measureTime() возвращает среднее время выполнения операции.

```
int main() {
    std::ofstream outFile("results.txt");

    outFile << "N, AVL Max Depth, AVL Avg Insert Time, AVL Avg Remove Time, AVL Avg Search Time, AVL Avg Height, Treap Max Depth, Treap Avg Insert Time, Treap Avg Remove Time, Treap Avg Search Time, Treap Avg Height, AVL Max Height of Last Series, Treap Max Height of Last Series, AVL Avg Branch Height of Last Series, Treap Avg Branch Height of Last Series\n";

    for (int i = 10; i <= 18; ++i) {
        int N = 1 << i;
        std::cout << "Running tests for N = " << N << std::endl;

        double totalInsertAVL = 0, totalRemoveAVL = 0, totalSearchAVL = 0;
        double totalInsertTreap = 0, totalRemoveTreap = 0, totalSearchTreap = 0;

        int maxDepthAVL = 0, maxDepthTreap = 0;
        std::vector<int> allDepthsAVL, allDepthsTreap;

        for (int repeat = 0; repeat < 50; ++repeat) {
```

```
            std::vector<int> values(N);
            for (int i = 0; i < N; ++i) {
                values[i] = rand() % 10000;
            }

            AVLTree avl_tree;
            for (int v : values) {
                avl_tree.insert(v);
            }

            totalInsertAVL     +=     avl_tree.measureTime([&](int     val)
{ avl_tree.insert(val); }, 1000);
            totalRemoveAVL     +=     avl_tree.measureTime([&](int     val)
{ avl_tree.remove(val); }, 1000);
            totalSearchAVL     +=     avl_tree.measureTime([&](int     val)
{ avl_tree.search(val); }, 1000);

            maxDepthAVL = std::max(maxDepthAVL, avl_tree.getMaxDepth());
            std::vector<int> depthsAVL = avl_tree.getAllDepths();
            allDepthsAVL.insert(allDepthsAVL.end(),       depthsAVL.begin(),
depthsAVL.end());

            Treap treap;
            for (int v : values) {
                treap.insert(v);
            }

            totalInsertTreap      +=      treap.measureTime([&](int     val)
{ treap.insert(val); }, 1000);
            totalRemoveTreap      +=      treap.measureTime([&](int     val)
{ treap.remove(val); }, 1000);
            totalSearchTreap      +=      treap.measureTime([&](int     val)
{ treap.search(val); }, 1000);

            maxDepthTreap = std::max(maxDepthTreap, treap.getMaxDepth());
            std::vector<int> depthsTreap = treap.getAllDepths();
            allDepthsTreap.insert(allDepthsTreap.end(), depthsTreap.begin(),
depthsTreap.end());
        }

        double avgInsertAVL = totalInsertAVL / 50;
        double avgRemoveAVL = totalRemoveAVL / 50;
        double avgSearchAVL = totalSearchAVL / 50;

        double avgInsertTreap = totalInsertTreap / 50;
        double avgRemoveTreap = totalRemoveTreap / 50;
        double avgSearchTreap = totalSearchTreap / 50;

        double    avgHeightAVL    =    std::accumulate(allDepthsAVL.begin(),
allDepthsAVL.end(), 0.0) / allDepthsAVL.size();
        double   avgHeightTreap   =   std::accumulate(allDepthsTreap.begin(),
allDepthsTreap.end(), 0.0) / allDepthsTreap.size();
```

```
            // Для последней серии
            double avgHeightAVLLastSeries = 0.0;
            double avgHeightTreapLastSeries = 0.0;
            double avgBranchHeightAVLLastSeries = 0.0;
            double avgBranchHeightTreapLastSeries = 0.0;

            if (!allDepthsAVL.empty()) {
                avgHeightAVLLastSeries  =  std::accumulate(allDepthsAVL.begin(),
allDepthsAVL.end(), 0.0) / allDepthsAVL.size();

                // Среднее распределение высот веток для AVL
                avgBranchHeightAVLLastSeries = avgHeightAVLLastSeries; // Здесь
можно использовать дополнительные формулы для расчёта высоты веток, если нужно.
            }

            if (!allDepthsTreap.empty()) {
                avgHeightTreapLastSeries                                    =
std::accumulate(allDepthsTreap.begin(),    allDepthsTreap.end(),    0.0)    /
allDepthsTreap.size();

                // Среднее распределение высот веток для Treap
                avgBranchHeightTreapLastSeries = avgHeightTreapLastSeries; // Для
расчёта высоты веток можно также использовать дополнительные подходы.
            }

            outFile << N << ", "
                << maxDepthAVL << ", "
                << avgInsertAVL << ", "
                << avgRemoveAVL << ", "
                << avgSearchAVL << ", "
                << avgHeightAVL << ", "
                << maxDepthTreap << ", "
                << avgInsertTreap << ", "
                << avgRemoveTreap << ", "
                << avgSearchTreap << ", "
                << avgHeightTreap << ", "
                << avgHeightAVLLastSeries << ", "
                << avgHeightTreapLastSeries << ", "
                << avgBranchHeightAVLLastSeries << ", "
                << avgBranchHeightTreapLastSeries << "\n";
        }

        outFile.close();
        std::cout << "Results have been written to results.txt" << std::endl;

        return 0;
    }
```

- Основная функция
    - Вход: Основная логика для анализа производительности структур данных.

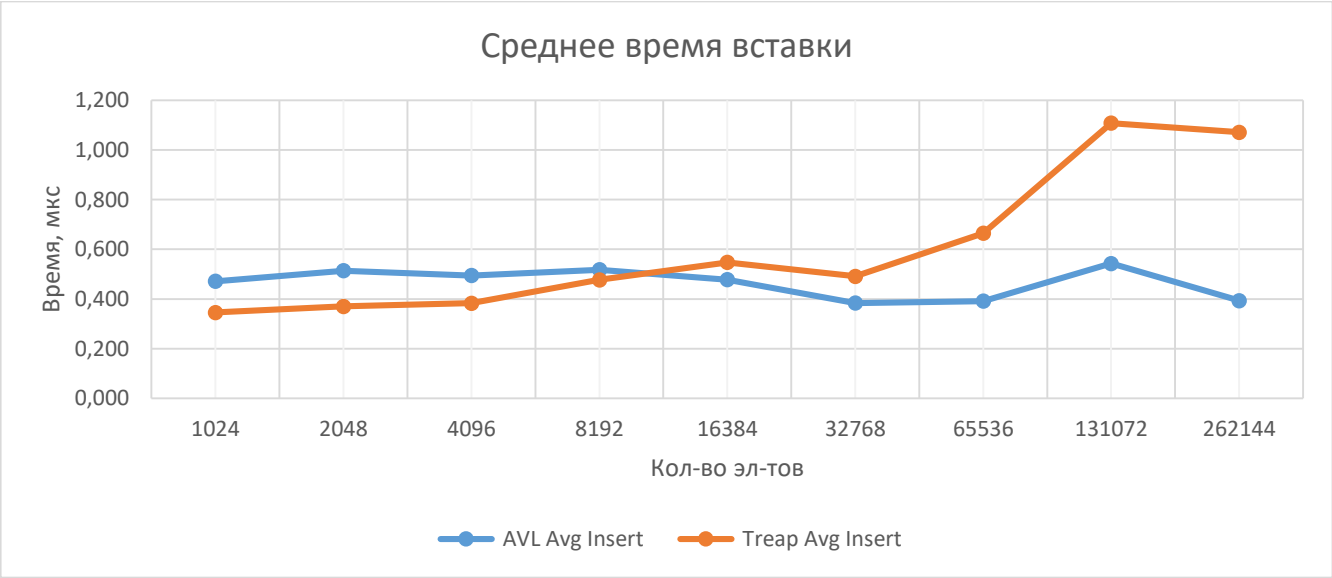
- Запускает цикл тестирования для разных значений N, от 10 до 18.
- В цикле создаёт структуры данных AVL и Treap, выполняет операции вставки, удаления, поиска и замеряет время.

- Работа: Тестирование производительности
  - Для каждого значения N генерируются случайные данные и выполняются операции вставки, удаления и поиска для обеих структур данных (AVL и Treap).
  - Время выполнения операций измеряется с помощью метода measureTime.
  - Также рассчитываются максимальная глубина дерева и среднее значение глубины всех узлов.
- Выход: Запись в файл.
  - Результаты тестов для каждого значения N записываются в файл results.txt, где для каждой итерации выводятся данные о максимальной глубине, среднем времени вставки, удаления и поиска, а также средняя высота и другие метрики для каждой структуры данных.

Заключение.

Результаты работы были представлены в виде таблицы значений, а затем интересующие данные были представлены в виде графиков

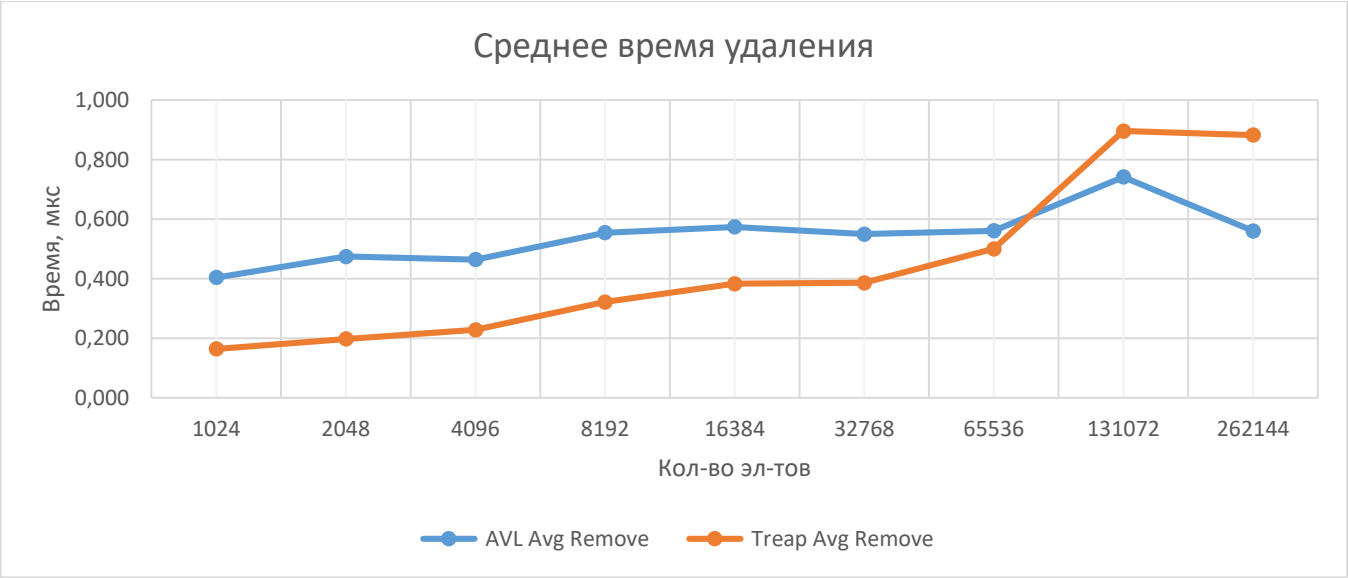
1. Анализ времени вставки
    - Среднее время вставки для AVL и Treap показывают разные тенденции с ростом N.
    - Время вставки для AVL деревьев варьируется, но наблюдается тенденция к уменьшению времени вставки по мере увеличения размера данных. Это может быть связано с тем, что при увеличении данных дерево AVL более эффективно балансируется, что снижает издержки на операцию вставки.
    - В среднем, AVL деревья показывают меньшее время вставки по сравнению с Treap. Однако важно отметить, что разница не всегда велика, и она становится более заметной на больших N, где время вставки для Treap увеличивается сильнее.
    - С увеличением размера N, время вставки постепенно увеличивается, при этом можно заметить, что для Treap этот рост более выражен, особенно на больших значениях N.



2. Анализ времени удаления
    - AVL и Treap показывают различные поведения в зависимости от размера данных N при операции удаления.
    - Время удаления в AVL деревьях растет с увеличением N, однако прирост не слишком велик, и время удаления в основном стабильное, несмотря на увеличение данных. Это связано с тем, что AVL использует строгую балансировку, и после удаления может потребоваться выполнить несколько вращений для поддержания баланса.
    - В Treap время удаления также растет с увеличением N, но значительно больше по сравнению с AVL. Это объясняется тем, что удаление в Treap не только требует

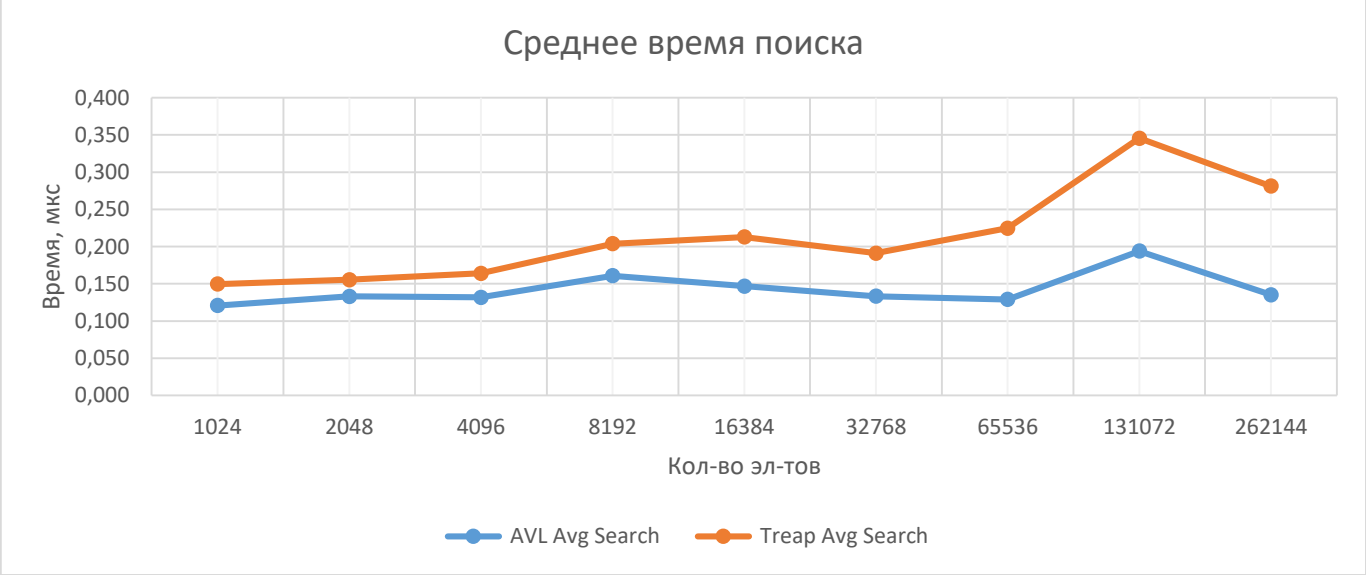
перестройки дерева, но и зависит от приоритетов узлов, что приводит к дополнительным затратам на перерасчет и балансировку структуры.

- Время удаления для AVL деревьев стабильно и в целом значительно ниже, чем для Treap.



3. Анализ времени поиска

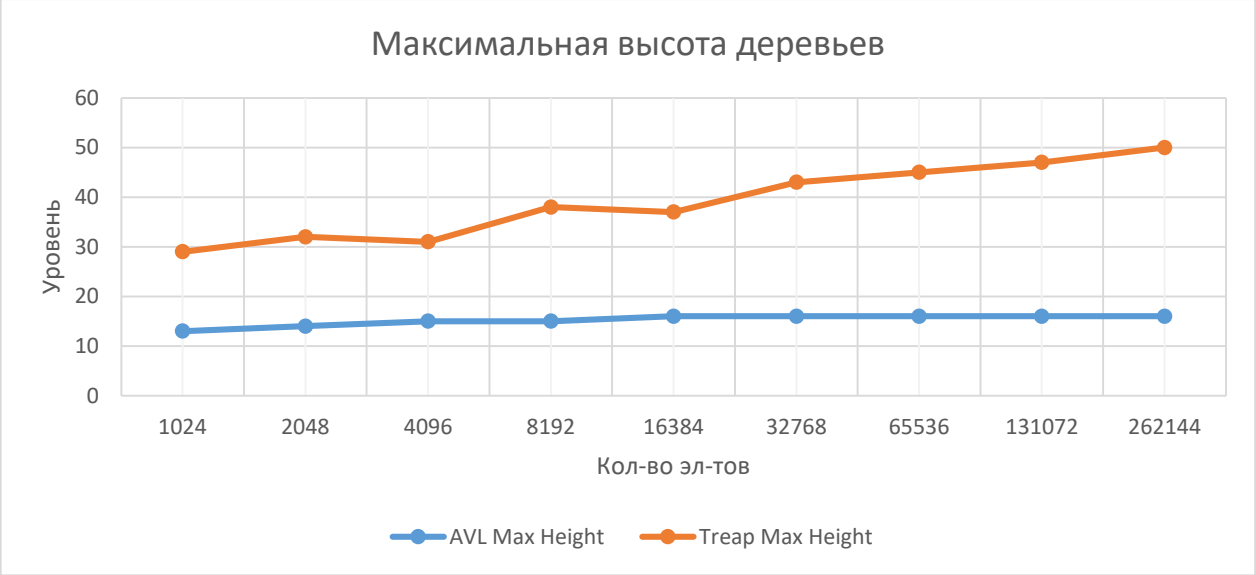
- время поиска увеличивается с ростом размера N для обеих структур, но Treap показывает более значительное увеличение времени поиска по сравнению с AVL.
- Treap зависит от случайных приоритетов, и время поиска в среднем может увеличиваться быстрее по мере роста структуры, так как балансировка зависит от случайных факторов, а не от строгих правил.



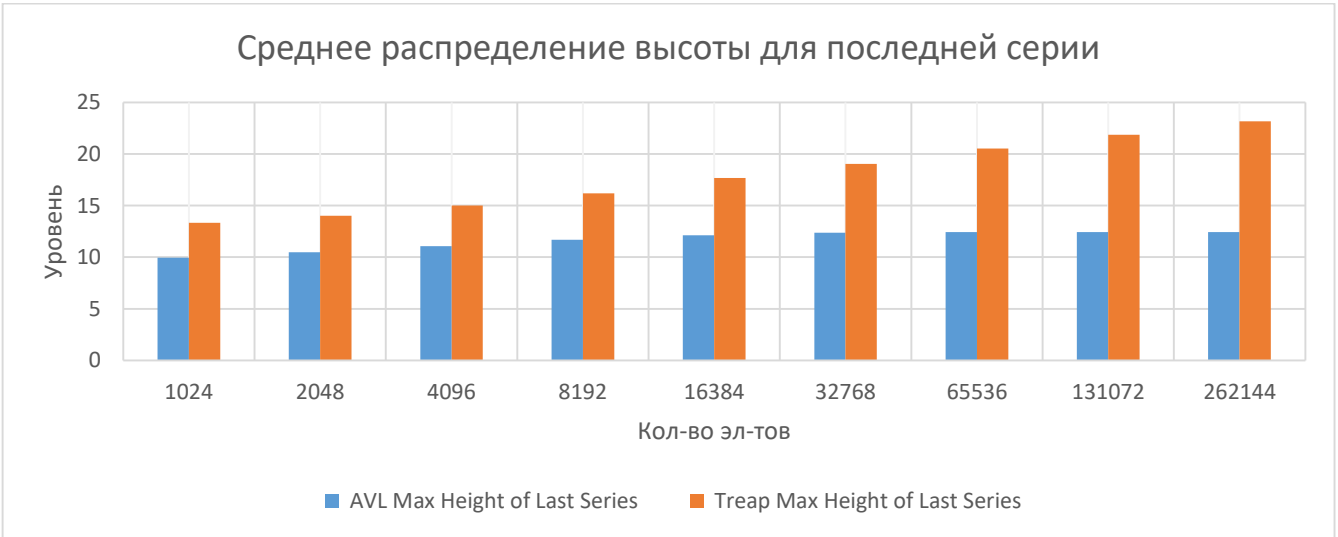
4. Анализ максимальной высоты

- AVL деревья, будучи сбалансированными, поддерживают свою высоту на достаточно низком уровне, независимо от увеличения количества элементов.

- Для Treap деревьев максимальная глубина существенно выше, чем для AVL деревьев. Это связано с тем, что Treap не гарантирует строгую сбалансированность, и при добавлении новых элементов может происходить более глубокое разрастание дерева.



5. Среднее распределение высоты последней серии
    - в случае AVL деревьев высота растет очень медленно, что является ожидаемым результатом, так как AVL дерево строго сбалансировано и само управляет высотой, поддерживая её на минимальном уровне, несмотря на количество элементов.
    - Treap менее сбалансирован, и из-за этого высота дерева растет быстрее, так как нет строгой гарантии сбалансированности, как в AVL деревьях.
    - AVL деревья обеспечивают стабильную и контролируемую среднюю максимальную высоту, что делает их лучшим выбором в ситуациях, когда важна предсказуемость и производительность.
    - Treap деревья, хотя и предлагают более гибкую структуру, страдают от более быстрого роста высоты, что может привести к снижению производительности при большом количестве элементов.



6. Среднее распределение высот веток последней серии
    - Рост высоты веток в AVL деревьях медленный и ограниченный, что подтверждает высокий уровень балансировки в этих деревьях.
    - Treap имеет более высокие ветки, что связано с его случайной балансировкой, которая может приводить к менее сбалансированным деревьям, что, в свою очередь, вызывает увеличение высоты веток.