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

Выполнил студент группы КС-36 Сары Кристина Ивановна

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/SaryKI\_36\_ALG.git

Приняли: Пысин Максим Дмитриевич

Краснов Дмитрий Олегович

Лобанов Алексей Владимирович

Крашенинников Роман Сергеевич

Дата сдачи: 24.02.2025

Оглавление

[Описание задачи. 3](#_Toc63548272)

[Описание метода/модели. 3](#_Toc63548273)

[Выполнение задачи. 4](#_Toc63548274)

[Заключение. 8](#_Toc63548275)

# Описание задачи.

**Задание:**

В рамках лабораторной работы необходимо изучить дерево поиска:

1. 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.

# Описание метода/модели.

**Теоретическая часть**

Декартово дерево (Treap) - это структура данных, сочетающая свойства бинарного дерева поиска и бинарной кучи. Каждый узел содержит два значения: ключ (упорядочивается как в бинарном дереве поиска) и приоритет (упорядочивается как в куче).

AVL-дерево - это сбалансированное по высоте бинарное дерево поиска, где для каждой вершины высота её двух поддеревьев различается не более чем на 1.

Основные отличия:

1. Декартово дерево использует рандомизацию для поддержания баланса
2. AVL-дерево поддерживает строгий баланс с помощью вращений
3. Ожидаемая высота Декартова дерева - O(log N), как и у AVL-дерева

**AVL-дерево**

**Определение:**  
AVL-дерево — это **самобалансирующееся** бинарное дерево поиска (БДП), в котором для каждой вершины высота её левого и правого поддеревьев отличается не более чем на 1. Балансировка достигается с помощью **вращений** (малого и большого) при нарушении условия сбалансированности.

**Свойства:**

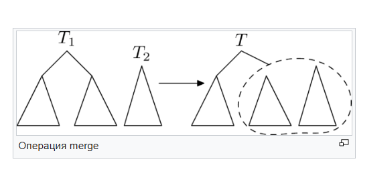
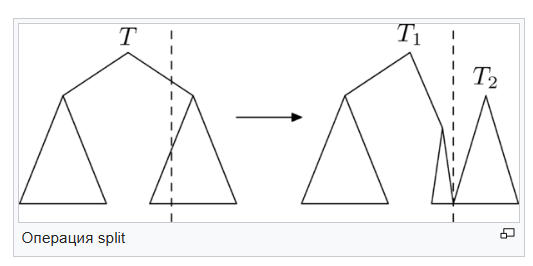
* **Гарантированная сбалансированность** (высота дерева всегда **~1.44 log₂N**).
* **Сложность операций** (вставка, удаление, поиск) — **O(log N)** в худшем случае.
* **Требует хранения баланс-фактора** (разница высот поддеревьев) в каждом узле.
* **Частые перебалансировки** при вставке/удалении, что может замедлить операции по сравнению с менее строгими структурами.

**Декартово дерево (Treap)**

**Определение:**  
Декартово дерево (Treap = Tree + Heap(куча)) — это **рандомизированная** структура данных, сочетающая свойства:

1. **Бинарного дерева поиска** (BST) по ключу.
2. **Двоичной кучи** (max-heap или min-heap) по приоритету (случайному числу).

**Свойства:**

* **Ожидаемая высота** — **O(log N)** (в среднем случае).
* **Сложность операций** (вставка, удаление, поиск) — **O(log N)** в среднем, но возможны выбросы до **O(N)** в худшем случае (крайне маловероятно).
* **Не требует балансировки** — баланс достигается за счёт случайных приоритетов.
* **Поддерживает быстрые операции split и merge** за **O(log N)**, что позволяет эффективно реализовывать объединение и разделение деревьев.
* 
* 

**Применение:**  
Используется в задачах, где важна **простота реализации** и **эффективное объединение/разделение** множеств (например, в алгоритмах на строках, персистентных структурах данных).

**2. Сравнение сложности операций**

|  |  |  |  |
| --- | --- | --- | --- |
| **Операция** | **AVL-дерево (худший случай)** | **Декартово дерево (средний случай)** | **Декартово дерево (худший случай)** |
| **Вставка** | O(log N) | O(log N) | O(N) (крайне редко) |
| **Удаление** | O(log N) | O(log N) | O(N) (крайне редко) |
| **Поиск** | O(log N) | O(log N) | O(N) (крайне редко) |
| **Split/Merge** | Не поддерживается | O(log N) | O(N) (крайне редко) |

**Ключевые отличия:**

1. **Балансировка:**
   * AVL требует **строгой балансировки** (разница высот ≤ 1).
   * Treap **не балансируется явно**, баланс достигается вероятностно.
2. **Скорость операций:**
   * AVL гарантирует **стабильное O(log N)**.
   * Treap в **среднем O(log N)**, но возможны редкие выбросы.
3. **Дополнительные операции:**
   * AVL не поддерживает эффективное **объединение/разделение**.
   * Treap позволяет делать **split и merge** за O(log N).
4. **Память:**
   * AVL хранит **баланс-фактор**.
   * Treap хранит **приоритет**.
5. **Реализация:**
   * AVL сложнее из-за **множества случаев вращений**.
   * Treap проще, так как баланс **автоматически поддерживается** приоритетами.

**Методика эксперимента**

Для сравнения были проведены серии тестов:

* 50 повторений для каждого N, где N = 2^i, i от 10 до 18
* Для каждого теста:
  1. Генерировались N случайных значений
  2. Заполнялись оба дерева в одинаковом порядке
  3. Замерялась максимальная глубина полученных деревьев
  4. Проводились 1000 операций вставки с замером времени
  5. Проводились 1000 операций удаления с замером времени
  6. Проводились 1000 операций поиска с замером времени
  7. Замерялись глубины всех веток дерева

# Выполнение задачи.

Программа реализована на языке C++.

**Код:**

**#include <iostream>**

**#include <fstream>**

**#include <algorithm>**

**#include <chrono>**

**#include <vector>**

**#include <random>**

**#include <numeric>**

**class AVLTree {**

**public:**

**struct Node {**

**int key;**

**Node\* left;**

**Node\* right;**

**int height;**

**Node(int key) : key(key), left(nullptr), right(nullptr), height(1) {}**

**};**

**AVLTree() : 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 if (key > node->key) node->right = insert(node->right, key);**

**else return node;**

**node->height = 1 + std::max(getHeight(node->left), getHeight(node->right));**

**int balance = getBalance(node);**

**if (balance > 1 && key < node->left->key)**

**return rightRotate(node);**

**if (balance < -1 && key > node->right->key)**

**return leftRotate(node);**

**if (balance > 1 && key > node->left->key) {**

**node->left = leftRotate(node->left);**

**return rightRotate(node);**

**}**

**if (balance < -1 && key < node->right->key) {**

**node->right = rightRotate(node->right);**

**return leftRotate(node);**

**}**

**return node;**

**}**

**Node\* remove(Node\* root, int key) {**

**if (!root) return root;**

**if (key < root->key) root->left = remove(root->left, key);**

**else if (key > root->key) root->right = remove(root->right, key);**

**else {**

**if (!root->left) {**

**Node\* temp = root->right;**

**delete root;**

**return temp;**

**} else if (!root->right) {**

**Node\* temp = root->left;**

**delete root;**

**return temp;**

**}**

**Node\* temp = getMinValueNode(root->right);**

**root->key = temp->key;**

**root->right = remove(root->right, temp->key);**

**}**

**root->height = 1 + std::max(getHeight(root->left), getHeight(root->right));**

**int balance = getBalance(root);**

**if (balance > 1 && getBalance(root->left) >= 0)**

**return rightRotate(root);**

**if (balance > 1 && getBalance(root->left) < 0) {**

**root->left = leftRotate(root->left);**

**return rightRotate(root);**

**}**

**if (balance < -1 && getBalance(root->right) <= 0)**

**return leftRotate(root);**

**if (balance < -1 && getBalance(root->right) > 0) {**

**root->right = rightRotate(root->right);**

**return leftRotate(root);**

**}**

**return root;**

**}**

**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\* rightRotate(Node\* y) {**

**Node\* x = y->left;**

**Node\* T2 = x->right;**

**x->right = y;**

**y->left = T2;**

**y->height = 1 + std::max(getHeight(y->left), getHeight(y->right));**

**x->height = 1 + std::max(getHeight(x->left), getHeight(x->right));**

**return x;**

**}**

**Node\* leftRotate(Node\* x) {**

**Node\* y = x->right;**

**Node\* T2 = y->left;**

**y->left = x;**

**x->right = T2;**

**x->height = 1 + std::max(getHeight(x->left), getHeight(x->right));**

**y->height = 1 + std::max(getHeight(y->left), getHeight(y->right));**

**return y;**

**}**

**int getHeight(Node\* node) {**

**if (!node) return 0;**

**return node->height;**

**}**

**int getBalance(Node\* node) {**

**if (!node) return 0;**

**return getHeight(node->left) - getHeight(node->right);**

**}**

**Node\* getMinValueNode(Node\* node) {**

**Node\* current = node;**

**while (current && current->left) current = current->left;**

**return current;**

**}**

**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);**

**}**

**};**

**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);**

**}**

**};**

**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;**

**}**

График зависимости:

* 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.**

1. **График зависимости среднего времени вставки от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.**

* Время вставки в **AVL-дерево** растет **медленнее** с увеличением N.
* Время вставки в **Treap** также имеет **логарифмический рост**, но с **большим разбросом**.
* При **малых N (1024–16384)** разница незначительна.
* При **больших N (131072–262144)** AVL показывает **более стабильное** время.

1. **AVL-дерево**:
   * Гарантированная балансировка (**высота ~1.44 log₂N**).
   * Вставка требует **O(log N) операций + возможные вращения**, но вращения незначительно влияют на общее время.
2. **Treap**:
   * **Средний случай O(log N)**, но возможны редкие **выбросы до O(N)** из-за неудачного распределения приоритетов.
   * **Не требует балансировки**, но при большом N **вероятность дисбаланса** увеличивается.

**Вывод:**

AVL-дерево **предсказуемее** на больших данных, Treap **чуть медленнее** из-за вероятностного баланса.

2) Среднее время поиска

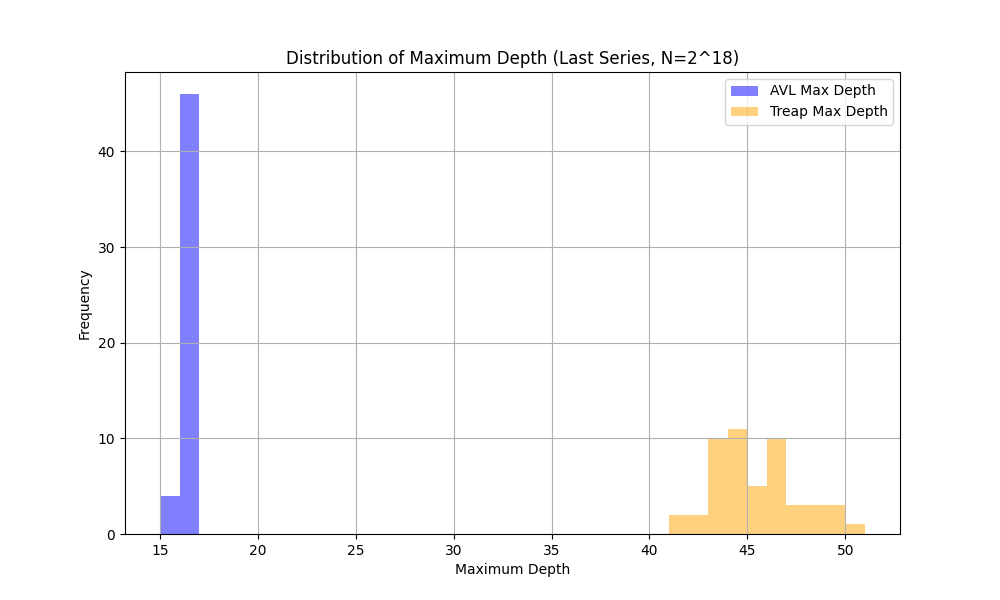
• время поиска увеличивается с ростом размера N для обеих структур, но Treap показывает более значительное увеличение времени поиска по сравнению с AVL.

• 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.

**3) Среднее время 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

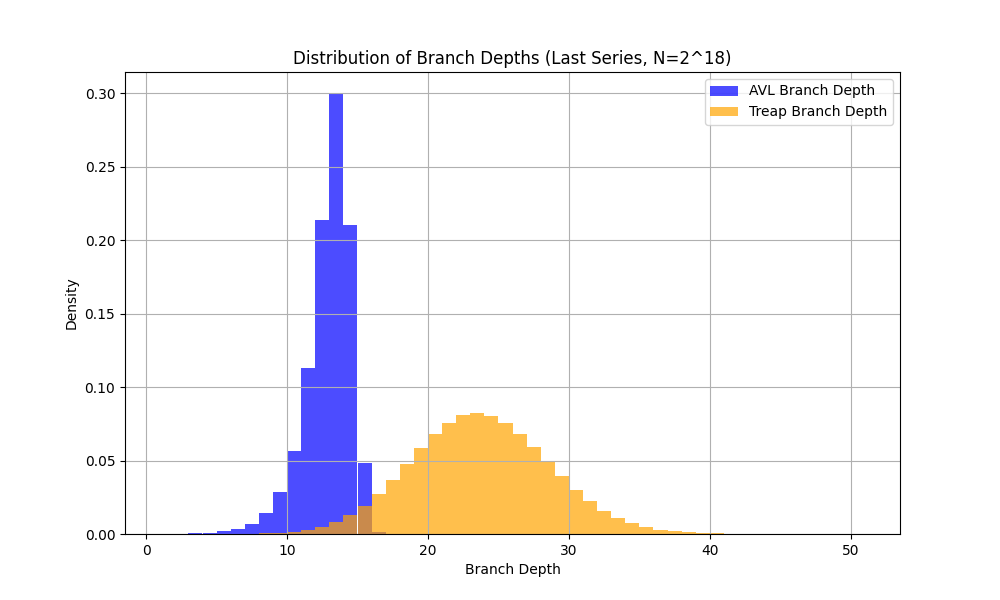
Гистограмму среднего распределения максимальной высоты для последней серии тестов для AVL и декартового дерева. **(Этот график — гистограмма, которая отображает распределение максимальной глубины (высоты) деревьев AVL и Treap, полученной в 50 тестах при N = 2^18.**

**Сравнение**

* AVL-дерево демонстрирует значительно лучшую сбалансированность по сравнению с Treap. Максимальная глубина AVL (15–16) почти в 3 раза меньше, чем у Treap (40–50), что делает операции (поиск, вставка, удаление) в AVL-дереве более предсказуемыми и быстрыми в худшем случае.
* Treap, несмотря на большую максимальную глубину, все еще остается эффективным в среднем случае. Однако в отдельных случаях случайные приоритеты могут привести к "вытягиванию" дерева, что увеличивает максимальную глубину.

Гистограмму среднего распределения высот веток в AVL дереве и для вашего варианта, для последней серии тестов

**(Этот график — гистограмма, которая отображает распределение глубин всех узлов (branch depths) в AVL-дереве и Treap для последней серии тестов (N = 2^18).)**

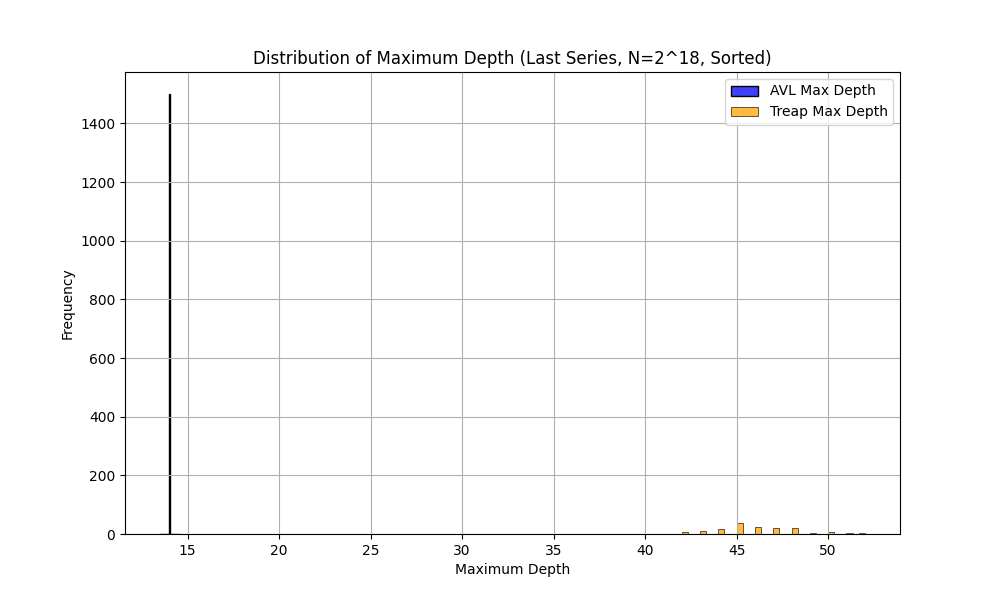
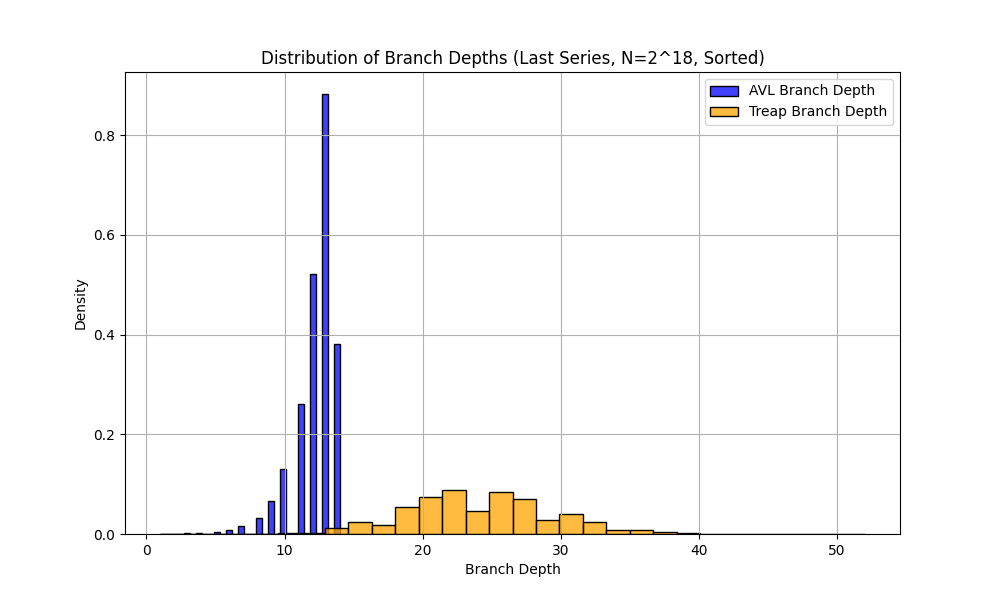
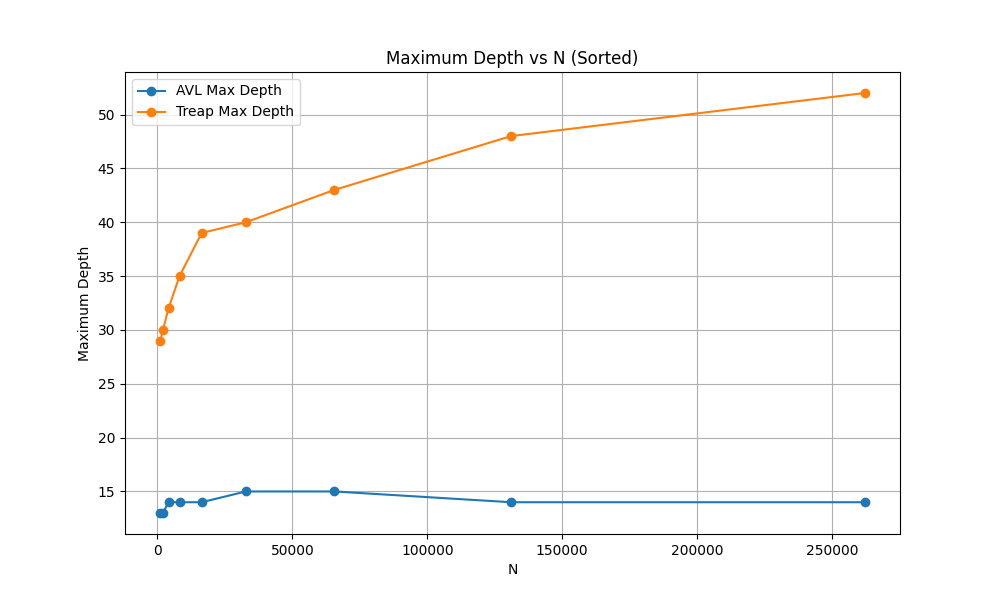
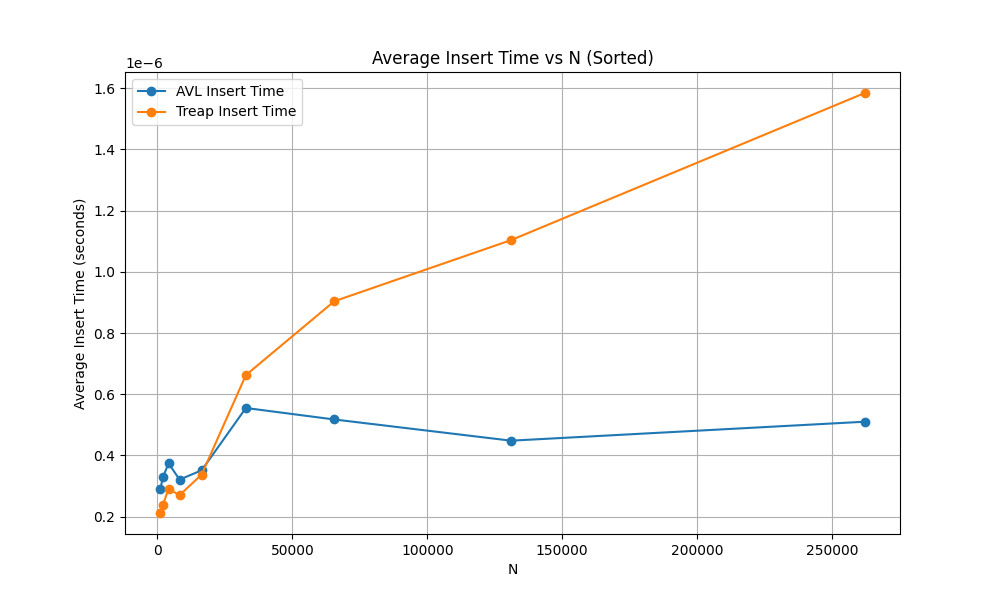
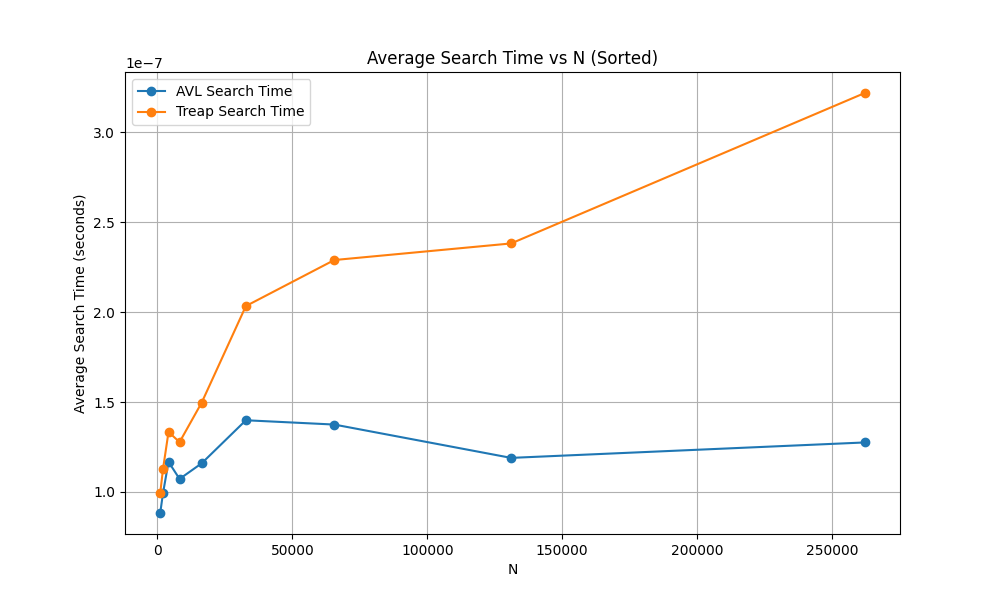
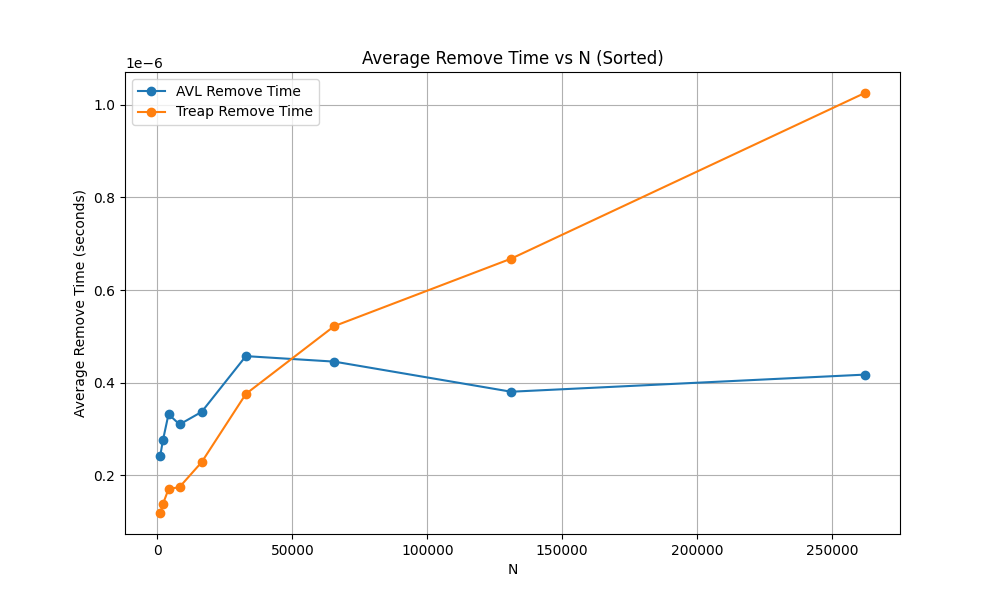


 Этот график показывает, как **распределяются узлы по глубинам** в AVL-дереве и Treap, что отражает их структуру и сбалансированность.

 AVL-дерево: Узлы сосредоточены на глубинах, близких к максимальной (10–15), что демонстрирует его компактность и равномерную сбалансированность. Практически нет узлов на больших глубинах, так как дерево минимизирует свою высоту.

 Treap: Узлы распределены более широко (1–50), с пиком на 15–20 и длинным "хвостом". Это показывает, что Treap в среднем сбалансирован,но из-за случайных приоритетов некоторые ветви могут быть значительно глубже, что приводит к большему разбросу глубин.

Для сортированных данных:



 **AVL-дерево**:

* Максимальная глубина осталась неизменной (15–16) для обоих наборов данных, что подтверждает строгую балансировку AVL-дерева.
* Сортированные данные приводят к более частому достижению глубины 16 (1400 против 200), так как дерево становится более "правым", но балансировка компенсирует это.
* Время операций остается, но при сортированных данных может быть больше поворотов, что слегка увеличивает константу в сложности вставки.

 **Treap**:

* Максимальная глубина (40–50) и распределение частот остались практически идентичными для случайных и сортированных данных, что подтверждает, что Treap не зависит от порядка вставки благодаря случайным приоритетам.
* Ожидаемая высота Treap —, но в отдельных тестах высота может быть больше (до 50), что мы и видим в обоих случаях.
* Время операций не изменилось, так как высота осталась в том же диапазоне.

**AVL-дерево**

* **Случайные данные**:
  + При случайной вставке ключи распределяются более равномерно между левым и правым поддеревьями, что минимизирует количество поворотов для балансировки.
* **Сортированные данные**:
  + При вставке отсортированных данных (например, по возрастанию) все ключи добавляются в правое поддерево, что приводит к временному дисбалансу на каждом шаге.
  + AVL-дерево выполняет повороты (левые или двойные левые) на каждом шаге, чтобы восстановить баланс. Это увеличивает количество поворотов по сравнению со случайными данными.
  + Несмотря на большее количество поворотов, высота дерева остается (15–16), и время операций не увеличивается асимптотически. Однако константа в сложности вставки может быть выше из-за дополнительных поворотов.
* **Итог для AVL**:
  + Порядок данных не влияет на асимптотическую сложность, так как высота дерева остается логарифмической.
  + При сортированных данных может увеличиться время вставки из-за большего числа поворотов, но разница незначительна.

**2. Treap**

* **Случайные данные**:
  + При случайной вставке Treap полагается на случайные приоритеты для балансировки. в отдельных тестах высота может быть больше (40–50).
  + Операции выполняются за ожидаемое время, но в худшем случае (глубина 50) время может быть больше.
* **Сортированные данные**:
  + При вставке отсортированных данных Treap ведет себя так же, как при случайных данных, потому что балансировка определяется случайными приоритетами, а не порядком ключей.
  + Если бы приоритеты не были случайными (например, всегда возрастали или убывали), Treap выродился бы в цепочку (глубина O(N). Но случайные приоритеты гарантируют, что даже при сортированных данных высота остается
  + Максимальная глубина (40–50) и распределение частот остались схожими, что подтверждает, что порядок данных не влияет на Treap.
* **Итог для Treap**:
  + Порядок данных не влияет на производительность Treap, так как балансировка определяется случайными приоритетами.
  + Ожидаемая сложность операций остается O(log⁡N)O(\log N)O(logN), и максимальная глубина в обоих случаях варьируется в одном диапазоне (40–50).

# Заключение.

1. Временные характеристики:
   * Операции в AVL-дереве выполняются за гарантированное O(log N) время
   * В Декартовом дереве среднее время также O(log N), но возможны выбросы
2. Структурные характеристики:
   * AVL-дерево поддерживает строгий баланс, что обеспечивает предсказуемую высоту
   * Декартово дерево в среднем имеет сравнимую высоту, но возможны отклонения
3. Практическое применение:
   * AVL-дерево предпочтительнее, когда критично гарантированное время операций
   * Декартово дерево проще в реализации и эффективнее для задач, требующих объединения/разделения деревьев

**Выводы**

1. Обе структуры данных демонстрируют логарифмическую сложность операций
2. AVL-дерево обеспечивает более стабильное время работы за счет строгого баланса
3. Декартово дерево проще в реализации и показывает сравнимую производительность в среднем случае
4. Для приложений, где важна предсказуемость, следует выбирать AVL-дерево
5. Для задач, где важна простота реализации и возможность объединения деревьев, предпочтительнее Декартово дерево