Санкт-Петербургский политехнический университет Петра Великого
Институт машиностроения, материала и транспорта
Высшая школа автоматизации и робототехники

Курсовая работа

Дисциплина: Объектно-ориентированное программирование

Тема: АВЛ-дерево

Выполнил студент гр. 3331506/20102 Бахвалов П.А.

Преподаватель Ананьевский М. С.

Санкт-Петербург

2025

Оглавление

Введение

АВЛ-деревом называется такое дерево поиска, в котором для любого его узла высоты левого и правого поддеревьев отличаются не более, чем на 1. Эта структура данных разработана советскими учеными Адельсон-Вельским Георгием Максимовичем и Ландисом Евгением Михайловичем в 1962 году. Аббревиатура АВЛ соответствует первым буквам фамилий этих ученых.

Данная работа содержит описание алгоритма, его программный код на языке C++, а также анализ алгоритма. При работе алгоритма реализуются добавление, удаление и поиск минимального ключа, а также балансировка дерева. Выполнен анализ вычислительной сложности алгоритма, включая временную сложность для основных операций и линейное использование памяти.

Описание алгоритма

AVL-деревья (сбалансированные двоичные деревья поиска) используются в задачах, где требуется эффективный поиск, вставка и удаление элементов с гарантированным временем работы $O(\log_n)$. Они находят применение в базах данных и файловых системах для индексации и ускорения поиска, а также в хранении отсортированных данных с быстрым доступом. В языках программирования AVL-деревья (или их аналоги, такие как красно-черные деревья) применяются для реализации ассоциативных массивов и словарей, где важен порядок элементов.

В геоинформационных системах (GIS) и компьютерной графике AVL-деревья помогают в поиске ближайших соседей и обработке пространственных данных. Они также используются в сетевых алгоритмах и маршрутизации, например, для хранения таблиц маршрутизации с быстрым поиском. В реальных-time системах, где критична предсказуемость времени выполнения операций, AVL-деревья предпочтительнее хеш-таблиц, так как гарантируют логарифмическое время работы даже в худшем случае.

Реализация алгоритма

Алгоритм был реализован при помощи языка программирования C++, полностью представлен в приложении 1. Узел дерева представлен структурой node, полями которой являются значение ключа в узле, высота дерева, указатель на структуру node для левой и правой ветви:

```
struct node // структура для представления узлов дерева
{
    int key;
    unsigned char height;
    node* left;
    node* right;
    node(int k) { key = k; left = right = 0; height = 1; }
};
```

Создан класс tree, содержащий конструктор без параметров, который создаёт корень дерева root. Класс также содержит методы основных операций над деревьями и методы вспомогательных операций.

```
class tree{

public:
    node *root;
    node* turnright(node* p);
    node* turnleft(node* q);
    node* findmin(node* p);
    node* removemin(node* p);
    node* remove(node* p, int k);
    node* search(node* p, int k);
    node* insert(node* p, int k);
    node* balance(node* p);
    tree()
    {
        root=new node;
        root->left=0;
        root->right=0;
        root->key=0;
        root->height=1;
    };
private:
    unsigned short height(node* p);
    int8_t bfactor(node* p);
    void realheight(node* p);
};
```

Отсутствие узлов слева или справа будем обнаруживать при помощи нулевого указателя в поле left и right соответственно. Для правильной работы программы необходимо реализовать функцию height:

```
unsigned char height(node* p)
{
    return p ? p->height : 0;
}
```

Если на вход подан отсутствующий узел, то она возвращает 0, иначе в поле height узла записывает высоту дерева с корнем в узле. Функция bfactor возвращает разницу между высотой правой и левой ветви. По свойству AVL дерева он может принимать значения -1, 0, 1.

```
int bfactor(node* p)
{
    return height(p->right) - height(p->left);
}
```

Следующая функция восстанавливает корректное значение поля height заданного узла:

```
void fixheight(node* p)
{
    unsigned char hl = height(p->left);
    unsigned char hr = height(p->right);
    p->height = (hl > hr ? hl : hr) + 1;
}
```

Заметим, что все три функции являются нерекурсивными.

В процессе добавления или удаления узлов в АВЛ-дереве возможно возникновение ситуации, когда balance factor некоторых узлов оказывается равными 2 или -2, т.е. возникает расбалансировка поддерева. Для выправления ситуации применяются повороты вокруг узлов дерева (рис. 1).

Рисунок 1 - Поворот влево

Код, реализующий правый и левый поворот, выглядит следующим образом (каждая функция, изменяющая дерево, возвращает новый корень полученного дерева):

```
node* rotateright(node* p) // правый поворот вокруг p
{
    node* q = p->left;
    p->left = q->right;
    q->right = p;
    fixheight(p);
    fixheight(q);
    return q;
}

node* rotateleft(node* q) // левый поворот вокруг q
{
    node* p = q->right;
    q->right = p->left;
    p->left = q;
    fixheight(q);
    fixheight(p);
    return p;
}
```

Код, выполняющий балансировку, сводится к проверке условий и выполнению поворотов:

```
node* balance(node* p) // балансировка узла p
{
    fixheight(p);
    if (bfactor(p) == 2)
    {
        if (bfactor(p->right) < 0)
            p->right = rotateright(p->right);
        return rotateleft(p);
    }
    if (bfactor(p) == -2)
    {
        if (bfactor(p->left) > 0)
            p->left = rotateleft(p->left);
        return rotateright(p);
    }
    return p; // балансировка не нужна
}
```

Если поддерево требует поворота влево для восстановления баланса, прежде следует проверить фактор сбалансированности его правого потомка. Если он перевешивает влево - повернуть его вправо, после чего сделать первоначальное левое вращение.

Если поддереву требуется поворот вправо для восстановления баланса, то сначала надо проверить фактор сбалансированности его левого потомка. Если он перевешивает вправо, то сначала повернуть его влево, а после выполнить первоначальное правое вращение.

Рисунок 2 демонстрирует эти правила. Начав с правого поворота вокруг узла C, дерево устанавливается в положение, при котором левый поворот вокруг узла A вернёт его в состояние баланса.

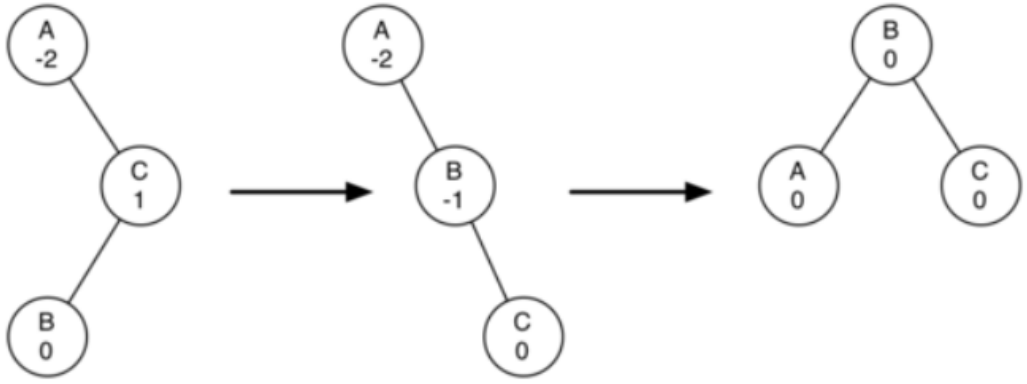

Рисунок 2 - Правый поворот с последующим левым вращением

Вставка нового ключа в АВЛ-дерево выполняется, по большому счету, так же, как это делается в простых деревьях поиска: спускаемся вниз по дереву, выбирая правое или левое направление движения в зависимости от результата сравнения ключа в текущем узле и вставляемого ключа. Единственное отличие заключается в том, что при возвращении из рекурсии (т.е. после того, как ключ вставлен либо в правое, либо в левое поддерево, и это дерево сбалансировано) выполняется балансировка текущего узла. Строго доказывается, что возникающий при такой вставке дисбаланс в любом узле по пути движения не превышает двух, а значит применение вышеописанной функции балансировки является корректным

```
node* insert(node* p, int k) // вставка ключа k в дерево с корнем p
{
    if (!p) return new node(k);
    if (k < p->key)
        p->left = insert(p->left, k);
    else
        p->right = insert(p->right, k);
    return balance(p);
}
```

Поиск ключей функцией search осуществляется проходом по дереву и сравнением искомого ключа и текущего, если ключ не найден, она сообщает об этом:

```
node* tree::search(node* p, int k) { // поиск ключа k дерева p
    while ((k != p->key) || ((p->left != nullptr) && (p->right != nullptr))) {
        if (k < p->key)p = p->left;
        if (k > p->key)p = p->right;
        if (k == p->key) return p;
        if ((p->left == nullptr) && (p->right == nullptr)) {
            std::cout << "can't find the key\n" << std::endl;
            return p;
        }
    }
}
```

Удаление ключей происходит функцией remove по следующей схеме: находится узел с заданным значением ключа k, затем если узел лист или не имеет правой ветви (указатель поля right нулевой), то в качестве текущего узла возвращается левый узел. Если правая ветвь у узла

с заданным значением ключа имеется, то ищется минимальное значение в правой ветви (необходимо спускаться по дереву, держась только левых узлов до конца), узел с текущим значением q удаляется, на его место вставляется минимальный элемент правой ветви и дерево балансируется.

```
node* findmin(node* p) // поиск узла с минимальным ключом в дереве p
{
    return p->left ? findmin(p->left) : p;
}

node* removemin(node* p) // удаление узла с минимальным ключом из дерева p
{
    if (p->left == 0)
        return p->right;
    p->left = removemin(p->left);
    return balance(p);
}

node* remove(node* p, int k) // удаление ключа k из дерева p
{
    if (!p) return 0;
    if (k < p->key)
        p->left = remove(p->left, k);
    else if (k > p->key)
        p->right = remove(p->right, k);
    else //  k == p->key
    {
        node* q = p->left;
        node* r = p->right;
        delete p;
        if (!r) return q;
        node* min = findmin(r);
        min->right = removemin(r);
        min->left = q;
        return balance(min);
    }
    return balance(p);
}
```

Анализ алгоритма

Время работы алгоритма:

Не для каждого набора ключей можно построить совершенное дерево, равно как и не для каждого набора ключей можно построить дерево Фибоначчи. Но эти деревья позволяют оценить диапазон возможных высот АВЛ-деревьев. Совершенное дерево является частным случаем идеально сбалансированного дерева, поэтому оно имеет минимально возможную высоту для данного количества узлов. Дерево Фибоначчи, напротив, имеет максимально возможную высоту для данного количества узлов, при условии, что сохраняются свойства АВЛ-дерева. Высота совершенного дерева вычисляется как $h = \log_2 (n + 1) + 1$ Высота дерева Фибоначчи вычисляется как $h < 1.44 \log_2 (n + 1) - 0.32$ Таким образом, учитывая оценку высоты совершенного дерева и оценку высоты дерева Фибоначчи получаем, что высота h АВЛ-дерева из n узлов находится в диапазоне $\log_2 (n + 1) \leq h < 1.44 \log_2 (n + 1) - 0.32$

Высота h АВЛ-дерева с n ключами лежит в диапазоне от $\log_2(n + 1)$ до $1.44 \log_2(n + 1) - 0.328$. Основные операции над двоичными деревьями поиска (поиск, вставка и удаление узлов) линейно зависят от его высоты, что гарантирует логарифмическую зависимость времени работы этих алгоритмов от числа ключей, хранимых в дереве.

Сложность по памяти алгоритма:

Реализация AVL-дерева характеризуется линейной сложностью по памяти O(n), где n представляет количество элементов в дереве. Эта характеристика обусловлена особенностями хранения структуры данных. Каждый узел AVL-дерева требует постоянного объема памяти O(1) для хранения ключа, двух указателей на дочерние узлы и информации о высоте поддерева или коэффициенте сбалансированности. Поскольку общее количество узлов равно n, совокупный объем памяти составляет O(n).

Проверка алгоритма

Проведём проверку времени работы функции вставки алгоритма. Результаты представлены на рисунке 3 и таблице 1. Код для проведения проверки представлен в приложении 2.

Таблица 1 – Зависимость времени на вставку от количества элементов массива

Кол-во	Вставка, нс
1000	1070,82
5000	1714,34
10000	2058,44
50000	2308,42
100000	2600,24
500000	3042,74

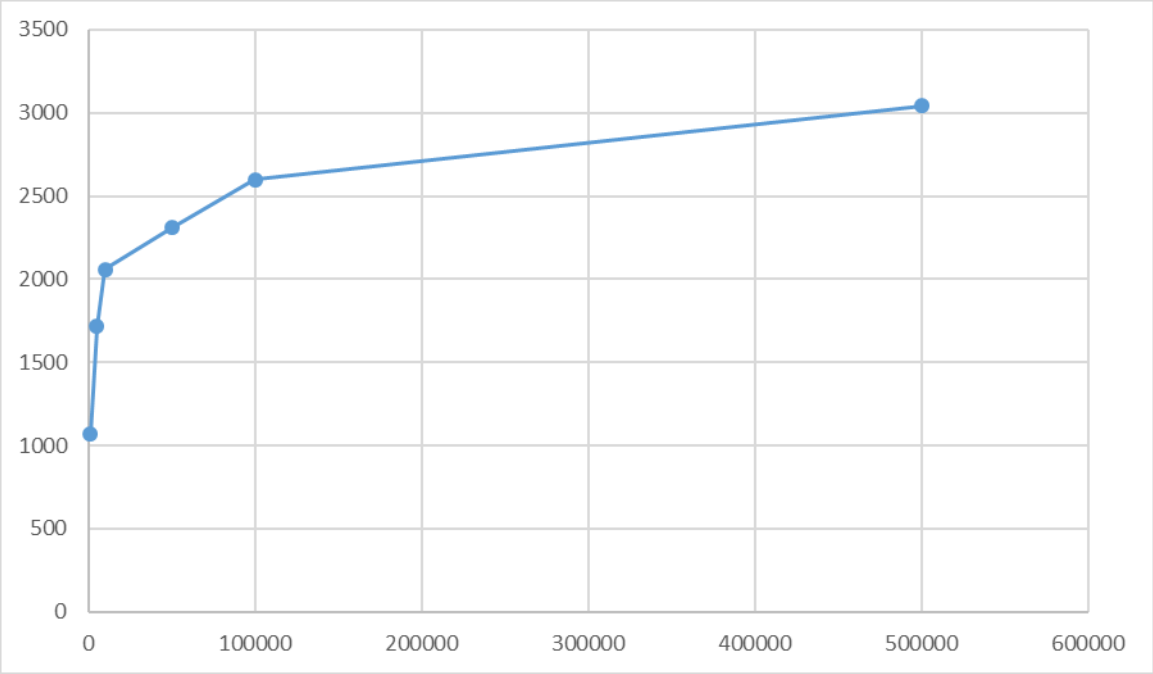

Рисунок 3 - Время на операцию вставка

Как мы видим график имеет форму логарифмической зависимости, что соответствует теоретическому O(n log n) времени на операцию.

Заключение

В ходе выполнения курсовой работы была исследована структура данных — АВЛ-дерево, реализован алгоритм его работы на языке C++, проведён анализ временной и пространственной сложности, а также выполнена проверка корректности работы алгоритма.

АВЛ-дерево — это мощный инструмент для работы с упорядоченными данными, особенно в задачах, где требуется частое выполнение операций поиска, вставки и удаления с гарантированным временем выполнения. Его применение оправдано в системах, обрабатывающих большие объёмы данных, где критична стабильность и предсказуемость производительности. Для задач с частыми изменениями данных и редкими поисками стоит рассмотреть альтернативные структуры, такие как красно-черные деревья или B-деревья.

Список литературы

1. *Адельсон-Вельский Г. М., Ландис Е. М.* Один алгоритм организации информации // Доклады АН СССР. — 1962. — Т. 146, № 2. — С. 263—266.

2. Н. Вирт, Алгоритмы и структуры данных. Пер. с англ. Ткачев Ф. В. – М.: ДМК Пресс, 2010. – 272 с.

Приложение 1

```cpp
#include <iostream>
#include <algorithm>
#include <chrono>
#include <random>
#include <vector>
#include <iomanip>

struct node {
    int key;
    unsigned short height;
    node* left;
    node* right;
    node(int k) : key(k), height(1), left(nullptr), right(nullptr) {}
};
class AVLTree {
private:
    node* root;

    // Вспомогательные функции
    unsigned short height(node* p) {
        return p ? p->height : 0;
    }

    int8_t bfactor(node* p) {
        return static_cast<int8_t>(height(p->right) - height(p->left));
    }

    void fixheight(node* p) {
        unsigned short hleft = height(p->left);
        unsigned short hright = height(p->right);
        p->height = (std::max(hleft, hright)) + 1;
    }

    // Балансировка
    node* rotateRight(node* p) {
        node* q = p->left;
        p->left = q->right;
        q->right = p;
        fixheight(p);
        fixheight(q);
        return q;
    }
```

```
node* rotateLeft(node* q) {
    node* p = q->right;
    q->right = p->left;
    p->left = q;
    fixheight(q);
    fixheight(p);
    return p;
}

node* balance(node* p) {
    fixheight(p);
    if (bfactor(p) == 2) {
        if (bfactor(p->right) < 0)
            p->right = rotateRight(p->right);
        return rotateLeft(p);
    }
    if (bfactor(p) == -2) {
        if (bfactor(p->left) > 0)
            p->left = rotateLeft(p->left);
        return rotateRight(p);
    }
    return p;
}
// Вставка и удаление
node* insert(node* p, int k) {
    if (!p) return new node(k);
    if (k < p->key)
        p->left = insert(p->left, k);
    else
        p->right = insert(p->right, k);
    return balance(p);
}

node* findmin(node* p) {
    return p->left ? findmin(p->left) : p;
}

node* removemin(node* p) {
    if (!p->left) return p->right;
    p->left = removemin(p->left);
    return balance(p);
}
```

```
node* remove(node* p, int k) {
    if (!p) return nullptr;
    if (k < p->key)
        p->left = remove(p->left, k);
    else if (k > p->key)
        p->right = remove(p->right, k);
    else {
        node* q = p->left;
        node* r = p->right;
        delete p;
        if (!r) return q;
        node* min = findmin(r);
        min->right = removemin(r);
        min->left = q;
        return balance(min);
    }
    return balance(p);
}

// Обходы дерева
void inorder(node* p) {
    if (p) {
        inorder(p->left);
        std::cout << p->key << " ";
        inorder(p->right);
    }
}
    void printTree(node* p, int indent = 0) {
        if (p) {
            if (p->right) printTree(p->right, indent + 4);
            if (indent) std::cout << std::setw(indent) << ' ';
            std::cout << p->key << "\n";
            if (p->left) printTree(p->left, indent + 4);
        }
    }

public:
    AVLTree() : root(nullptr) {}

    void insert(int k) {
        root = insert(root, k);
    }

    void remove(int k) {
        root = remove(root, k);
    }

    void printSorted() {
        inorder(root);
        std::cout << std::endl;
    }

    void print() {
        printTree(root);
    }
};
```

```cpp
int main() {
    AVLTree tree;
    int values[] = { 10, 20, 30, 40, 50, 25, 15, 5, 35, 45 };
    int n = sizeof(values) / sizeof(values[0]);

    std::cout << "Small tree demo:\n";
    std::cout << "Inserting values: ";
    for (int i = 0; i < n; i++) {
        std::cout << values[i] << " ";
        tree.insert(values[i]);
    }
    std::cout << "\n\n";

    std::cout << "Sorted values: ";
    tree.printSorted();
    std::cout << "\nTree structure:\n";
    tree.print();
    std::cout << "\n";

    return 0;
}
```

Приложение 2

```
#include <iostream>
#include <algorithm>
#include <chrono>
#include <random>
#include <vector>
#include <iomanip>
#include <cmath>
#include <numeric>
#include <iterator>

struct node {
    int key;
    unsigned short height;
    node* left;
    node* right;
    node(int k) : key(k), height(1), left(nullptr), right(nullptr) {}
};

class AVLTree {
private:
    node* root;

    // Вспомогательные функции
    unsigned short height(node* p) {
        return p ? p->height : 0;
    }

    int8_t bfactor(node* p) {
        return static_cast<int8_t>(height(p->right) - height(p->left));
    }

    void fixheight(node* p) {
        unsigned short hleft = height(p->left);
        unsigned short hright = height(p->right);
        p->height = (std::max(hleft, hright)) + 1;
    }

    // Балансировка
    node* rotateRight(node* p) {
        node* q = p->left;
        p->left = q->right;
        q->right = p;
        fixheight(p);
        fixheight(q);
        return q;
    }

    node* rotateLeft(node* q) {
        node* p = q->right;
        q->right = p->left;
        p->left = q;
        fixheight(q);
        fixheight(p);
        return p;
    }

    node* balance(node* p) {
        fixheight(p);
        if (bfactor(p) == 2) {
            if (bfactor(p->right) < 0)
                p->right = rotateRight(p->right);
```

```
        return rotateLeft(p);
    }
    if (bfactor(p) == -2) {
        if (bfactor(p->left) > 0)
            p->left = rotateLeft(p->left);
        return rotateRight(p);
    }
    return p;
}

// Вставка и удаление
node* insert(node* p, int k) {
    if (!p) return new node(k);
    if (k < p->key)
        p->left = insert(p->left, k);
    else
        p->right = insert(p->right, k);
    return balance(p);
}

node* findmin(node* p) {
    return p->left ? findmin(p->left) : p;
}

node* removemin(node* p) {
    if (!p->left) return p->right;
    p->left = removemin(p->left);
    return balance(p);
}

node* remove(node* p, int k) {
    if (!p) return nullptr;
    if (k < p->key)
        p->left = remove(p->left, k);
    else if (k > p->key)
        p->right = remove(p->right, k);
    else {
        node* q = p->left;
        node* r = p->right;
        delete p;
        if (!r) return q;
        node* min = findmin(r);
        min->right = removemin(r);
        min->left = q;
        return balance(min);
    }
    return balance(p);
}

// Обходы дерева
void inorder(node* p) {
    if (p) {
        inorder(p->left);
        std::cout << p->key << " ";
        inorder(p->right);
    }
}

void printTree(node* p, int indent = 0) {
    if (p) {
        if (p->right) printTree(p->right, indent + 4);
        if (indent) std::cout << std::setw(indent) << ' ';
        std::cout << p->key << "\n";
        if (p->left) printTree(p->left, indent + 4);
    }
```

```
    }

public:
    AVLTree() : root(nullptr) {}

    void insert(int k) {
        root = insert(root, k);
    }

    void remove(int k) {
        root = remove(root, k);
    }

    void printSorted() {
        inorder(root);
        std::cout << std::endl;
    }

    void print() {
        printTree(root);
    }
};

template<typename InputIt, typename OutputIt, typename Size, typename RNG>
void manual_sample(InputIt first, InputIt last, OutputIt out, Size n, RNG&& g) {
    std::vector<typename std::iterator_traits<InputIt>::value_type> pool(first,
last);
    std::shuffle(pool.begin(), pool.end(), g);

    for (Size i = 0; i < n && i < pool.size(); ++i) {
        *out++ = pool[i];
    }
}

void precise_performance_test() {
    std::vector<int> sizes = { 1000, 5000, 10000, 50000, 100000, 500000 };
    const int warmup_runs = 3;
    const int measured_runs = 5;
    const int search_tests = 10000;

    std::cout << "Precise Performance Analysis:\n";
    std::cout << std::setw(10) << "Size"
        << std::setw(15) << "Insert (ms)"
        << std::setw(15) << "Insert/n (ns)"
        << std::setw(15) << "Search (ns)"
        << std::setw(20) << "Theor O(log n)\n";

    std::random_device rd;
    std::mt19937 gen(rd());

    for (int size : sizes) {
        // Прогрев кэша
        for (int w = 0; w < warmup_runs; ++w) {
            AVLTree warmup_tree;
            std::vector<int> warmup_data(size);
            std::iota(warmup_data.begin(), warmup_data.end(), 1);
            std::shuffle(warmup_data.begin(), warmup_data.end(), gen);
            for (int val : warmup_data) {
                warmup_tree.insert(val);
            }
        }

        // Основные измерения
        double total_insert_time = 0;
        double total_search_time = 0;
```

```
        for (int r = 0; r < measured_runs; ++r) {
            AVLTree tree;
            std::vector<int> data(size);
            std::iota(data.begin(), data.end(), 1);
            std::shuffle(data.begin(), data.end(), gen);

            // Тест вставки
            auto insert_start = std::chrono::high_resolution_clock::now();
            for (int val : data) {
                tree.insert(val);
            }
            auto insert_end = std::chrono::high_resolution_clock::now();
            total_insert_time +=
std::chrono::duration_cast<std::chrono::nanoseconds>(insert_end -
insert_start).count();

            // Тест поиска с ручной выборкой
            std::vector<int> search_keys;
            manual_sample(data.begin(), data.end(), std::back_inserter(search_keys),
                search_tests, gen);

            auto search_start = std::chrono::high_resolution_clock::now();
            for (int key : search_keys) {
                // tree.search(key); // Раскомментировать после реализации search()
            }
            auto search_end = std::chrono::high_resolution_clock::now();
            total_search_time +=
std::chrono::duration_cast<std::chrono::nanoseconds>(search_end -
search_start).count();
        }

        // Расчет средних значений
        double avg_insert_time = total_insert_time / (measured_runs * size);
        double avg_search_time = total_search_time / (measured_runs * search_tests);
        double theoretical = 50 * log2(size);

        std::cout << std::setw(10) << size
            << std::setw(15) << std::fixed << std::setprecision(2) <<
total_insert_time / (measured_runs * 1e6)
            << std::setw(15) << std::fixed << std::setprecision(2) <<
avg_insert_time
            << std::setw(15) << std::fixed << std::setprecision(2) <<
avg_search_time
            << std::setw(20) << std::fixed << std::setprecision(2) << theoretical
            << "\n";
    }
}

int main() {
    precise_performance_test();
    return 0;
}
```