

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

Факультет цифровых технологий и химического инжиниринга

Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ
№ 6 ПО КУРСУ
«АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ»:

СТУДЕНТ группы КС-33

Костяева К.С.

Москва
2024

ОГЛАВЛЕНИЕ

ТЕОРИЯ	3
Декартово дерево	3
Операции декартового дерева	3
Объединение деревьев (Merge)	3
Разделение дерева (Split).....	4
ЗАДАНИЕ	6
ПРАКТИЧЕСКАЯ ЧАСТЬ.....	7
Код:.....	7
Результат работы программы:.....	14
Графики:	16
ВЫВОД	22

ТЕОРИЯ

Декартово дерево

Декартово дерево — это структура данных, которая сочетает в себе свойства двоичного дерева поиска и кучи. Декартово дерево в каждом узле помимо ключа, хранит так же приоритет узла, который отражает позицию элемента в такой структуре данных как куча (древовидная структура, у которой родитель дерева больше всех его потомков (или меньше)).

Для построения декартового дерева, нам потребуется:

- Множество ключей, это те значения, которые есть в наших исходных данных, по которым мы и хотим построить декартово дерево поиска. Одинаковые ключи следует хранить либо только в правом поддереве, либо только в левом.
- Множество приоритетов, по своей сути, случайная величина, которую мы можем как генерировать самостоятельно, так и брать, из поступающих нам данных, связанных с ключами. Основным ограничением является то, что они должны быть случайными. Одинаковых приоритетов стоит избегать, в идеале генерировать случайные числа от 0 до 1, но, если нужно, можно и случайное целочисленное число.

Операции декартового дерева

Для работы с деревом нам понадобятся две важные операции:

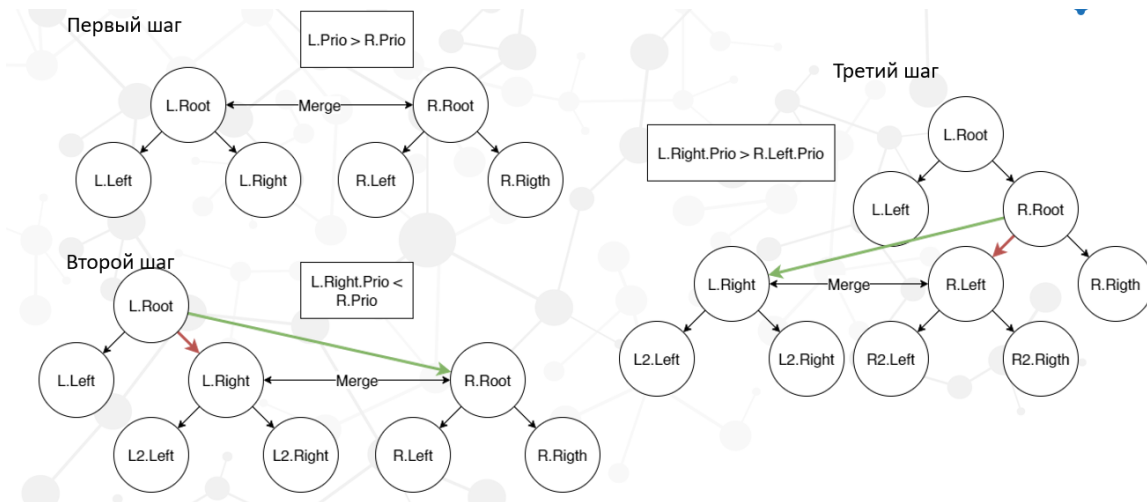
- Объединение деревьев (Merge) – операция, которая берет два декартовых дерева L - левое и R правое и получает в итоге новое декартово дерево, которое является их объединением.
- Разделение дерева (Split) – операция, которая берет одно декартово дерево и делит их на два по значению какого-либо ключа так, чтобы левое дерево было строго меньше ключа, а правое поддерево строго больше ключа.

Объединение деревьев (Merge)

Для работы этой операции у нас должно быть два поддерева L и R таких, чтобы каждый ключ из L был строго меньше любого ключа из R и наоборот, любой ключ из R был строго больше любого ключа из L.

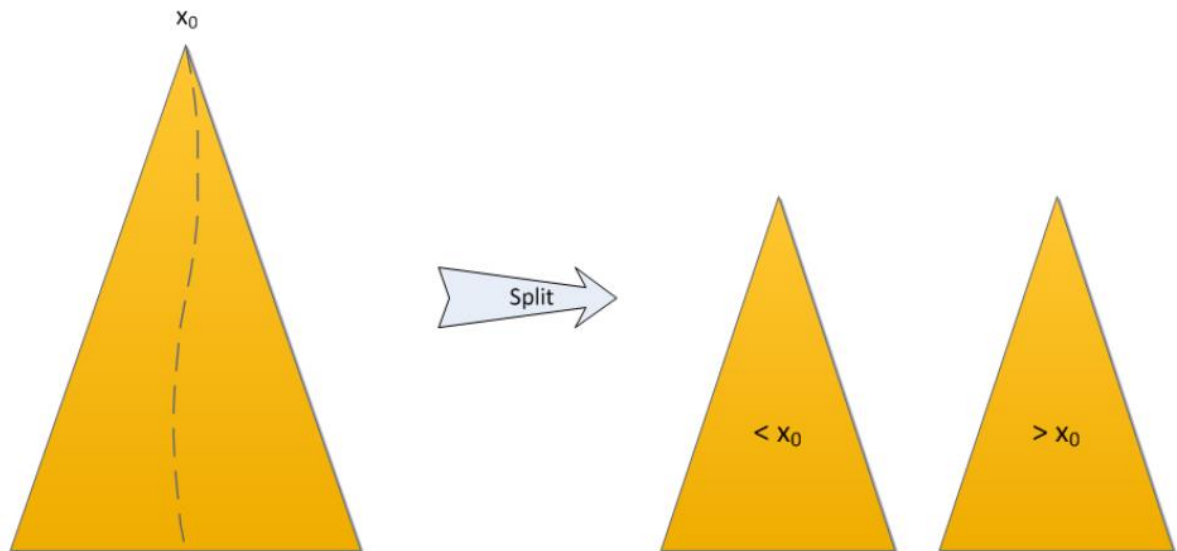
Как объединить два дерева:

1. Сравнить приоритеты корней L и R. Больший делаем корнем нового дерева.
2. Если корнем стал корень L, то все его левое поддерево остается на своем месте, так как оно строго меньше корня.
 - а. Поскольку правое R строго больше корня L по условию операции и должно оказаться справа от корня L, то для его вставки в правое поддерево L следует выполнить операцию объединения R и правого поддерева корня L.
3. Если корнем стал корень R, то все его правое поддерево остается на своем месте, так как оно строго больше корня.
 - а. Поскольку левое L строго меньше корня R по условию операции и должно оказаться слева от корня R, то оно конфликтует с левым поддеревом корня R, а значит следует выполнить операцию объединения L и левого поддерева корня R.



Разделение дерева (Split)

Теперь об операции Split. На вход ей поступает корректное декартово дерево T и некий ключ x_0 . Задача операции — разделить дерево на два так, чтобы в одном из них (L) оказались все элементы исходного дерева с ключами, меньшими x_0 , а в другом (R) — с большими. Никаких особых ограничений на дерево не накладывается.



Рассуждаем похожим образом. Где окажется корень дерева T ? Если его ключ меньше x_0 , то в L , иначе в R . Опять-таки, предположим для однозначности, что ключ корня оказался меньше x_0 .

Тогда можно сразу сказать, что все элементы левого поддерева T также окажутся в L — их ключи ведь тоже все будут меньше x_0 . Более того, корень T будет и корнем L , поскольку его приоритет наибольший во всем дереве. Левое поддерево корня полностью сохранится без изменений, а вот правое уменьшится — из него придется убрать элементы с ключами, большими x_0 , и вынести в дерево R . А остаток ключей сохранить как новое правое поддерево L .

Для работы операции, ей на вход подается декартово дерево T и ключ x по которому происходит деление дерева на две части, L — которая строго меньше чем x , и R — которая строго больше чем x .

Как же их разделить?

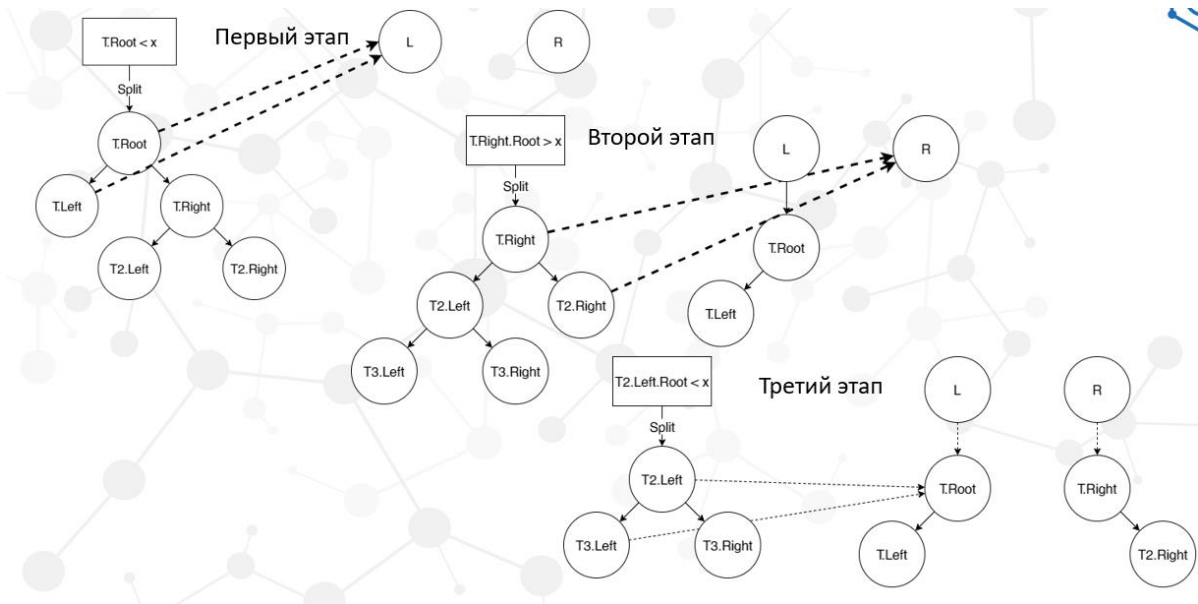
1. Сравниваем корень дерева со значением ключа разделения.
2. Если корень T меньше ключа по которому происходит деление, то именно этот узел и будет являться корнем результата деления L , а все левое поддерево будет являться частью L , так

как оно меньше T, которая меньше x.

- а. Поскольку правое поддерево T строго больше чем корень, то следует повторить процесс разбиения по x на всем правом поддереве, так как нам нужно отделить узлы, которые больше x.

3. Если корнем T больше ключа по которому происходит деление, то именно этот узел и будет является корнем результата деления R, а все правое поддерево будет является частью R, так как оно больше T, которое больше x.

- а. Поскольку левое поддерево L строго меньше чем корень, то следует повторить процесс разбиения по x на всем левом поддереве, так как нам нужно отделить узлы, которые меньше x.



ЗАДАНИЕ

вариант 1

Задание на лабораторную #7.

В рамках лабораторной работы необходимо изучить одно из двух деревьев поиска:

1. Декартово дерево (<https://habr.com/ru/post/101818/>)
2. Рандомизированное дерево (<https://habr.com/ru/post/145388/>)

Для этого его потребуется реализовать и сравнить в работе с реализованным ранее AVL-деревом. Для анализа работы алгоритма понадобится 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.

ПРАКТИЧЕСКАЯ ЧАСТЬ

Код:

язык программирования C++

```
#include <iostream>
#include <fstream>
#include <random>
#include <chrono>
#include <vector>
#include <algorithm>
#include <queue>
#include <stack>
#include <cmath>
#include <numeric> // для std::iota
#include <cassert>

using namespace std;
using namespace std::chrono;

struct AVLNode {
    int key;
    AVLNode* left;
    AVLNode* right;
    int height;

    AVLNode(int k) : key(k), left(nullptr), right(nullptr), height(1) {}
};

struct TreapNode {
    int key; // ключ (соблюдает свойства BST)
    int priority; // приоритет (соблюдает свойства кучи)
    TreapNode* left;
    TreapNode* right;

    TreapNode(int k, int p) : key(k), priority(p), left(nullptr), right(nullptr) {}
};

class AVLTree {
private:
    AVLNode* root;

    int height(AVLNode* node) {
        return node ? node->height : 0;
    }

    int balanceFactor(AVLNode* node) {
        return height(node->right) - height(node->left);
    }

    void updateHeight(AVLNode* node) {
        int hl = height(node->left);
        int hr = height(node->right);
        node->height = (hl > hr ? hl : hr) + 1;
    }

    AVLNode* rotateRight(AVLNode* y) {
        AVLNode* x = y->left;
        y->left = x->right;
        x->right = y;
        updateHeight(y);
        updateHeight(x);
        return x;
    }

    AVLNode* rotateLeft(AVLNode* x) {
        AVLNode* y = x->right;
        x->right = y->left;
```

```

        y->left = x;
        updateHeight(x);
        updateHeight(y);
        return y;
    }

AVLNode* balance(AVLNode* node) {
    updateHeight(node);
    int bf = balanceFactor(node);

    if (bf == 2) {
        if (balanceFactor(node->right) < 0)
            node->right = rotateRight(node->right);
        return rotateLeft(node);
    }
    if (bf == -2) {
        if (balanceFactor(node->left) > 0)
            node->left = rotateLeft(node->left);
        return rotateRight(node);
    }
    return node;
}

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

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

    return balance(node);
}

AVLNode* findMin(AVLNode* node) {
    return node->left ? findMin(node->left) : node;
}

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

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

    if (key < node->key)
        node->left = remove(node->left, key);
    else if (key > node->key)
        node->right = remove(node->right, key);
    else {
        AVLNode* l = node->left;
        AVLNode* r = node->right;
        delete node;
        if (!r) return l;
        AVLNode* min = findMin(r);
        min->right = removeMin(r);
        min->left = l;
        return balance(min);
    }
    return balance(node);
}

bool search(AVLNode* node, int key) {
    if (!node) return false;
    if (key == node->key) return true;

```



```

        return key < node->key ? search(node->left, key) : search(node->right, key);
    }

    void getDepths(AVLNode* node, int currentDepth, vector<int>& depths) {
        if (!node) return;
        if (!node->left && !node->right) {
            depths.push_back(currentDepth);
            return;
        }
        getDepths(node->left, currentDepth + 1, depths);
        getDepths(node->right, currentDepth + 1, depths);
    }

    void clear(AVLNode* node) {
        if (!node) return;
        clear(node->left);
        clear(node->right);
        delete node;
    }

public:
    AVLTree() : root(nullptr) {}
    ~AVLTree() { clear(root); }

    void insert(int key) { root = insert(root, key); }
    void remove(int key) { root = remove(root, key); }
    bool search(int key) { return search(root, key); }

    int maxDepth() {
        if (!root) return 0;
        queue<pair<AVLNode*, int>> q;
        q.push({ root, 1 });
        int maxDepth = 0;

        while (!q.empty()) {
            auto current = q.front();
            q.pop();
            maxDepth = max(maxDepth, current.second);

            if (current.first->left)
                q.push({ current.first->left, current.second + 1 });
            if (current.first->right)
                q.push({ current.first->right, current.second + 1 });
        }
        return maxDepth;
    }

    vector<int> getAllDepths() {
        vector<int> depths;
        getDepths(root, 1, depths);
        return depths;
    }
};

class Treap {
private:
    TreapNode* root;

    void split(TreapNode* node, int key, TreapNode*& left, TreapNode*& right) {
        if (!node) {
            left = right = nullptr;
            return;
        }

        if (node->key < key) {
            split(node->right, key, node->right, right);
            left = node;
        }
    }

```

```

        else {
            split(node->left, key, left, node->left);
            right = node;
        }
    }

TreapNode* merge(TreapNode* left, TreapNode* right) {
    if (!left) return right;
    if (!right) return left;

    if (left->priority > right->priority) {
        left->right = merge(left->right, right);
        return left;
    }
    else {
        right->left = merge(left, right->left);
        return right;
    }
}

//Вставляет узел как в BST, но затем проверяет приоритеты
TreapNode* insert(TreapNode* node, int key, int priority) {
    if (!node) return new TreapNode(key, priority);

    if (key < node->key) {
        node->left = insert(node->left, key, priority);
        if (node->left->priority > node->priority) {
            node = rotateRight(node);
        }
    }
    else {
        node->right = insert(node->right, key, priority);
        if (node->right->priority > node->priority) {
            node = rotateLeft(node);
        }
    }

    return node;
}

TreapNode* rotateRight(TreapNode* y) {
    TreapNode* x = y->left;
    y->left = x->right;
    x->right = y;
    return x;
}

TreapNode* rotateLeft(TreapNode* x) {
    TreapNode* y = x->right;
    x->right = y->left;
    y->left = x;
    return y;
}

//Если найден узел – его левое и правое поддеревья сливаются с помощью merge
TreapNode* remove(TreapNode* node, int key) {
    if (!node) return nullptr;

    if (key < node->key) {
        node->left = remove(node->left, key);
    }
    else if (key > node->key) {
        node->right = remove(node->right, key);
    }
    else {
        TreapNode* temp = node;
        node = merge(node->left, node->right);
        delete temp;
    }
}

```

```

    }
    return node;
}

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

void getDepths(TreapNode* node, int currentDepth, vector<int>& depths) {
    if (!node) return;
    if (!node->left && !node->right) {
        depths.push_back(currentDepth);
        return;
    }
    getDepths(node->left, currentDepth + 1, depths);
    getDepths(node->right, currentDepth + 1, depths);
}

void clear(TreapNode* node) {
    if (!node) return;
    clear(node->left);
    clear(node->right);
    delete node;
}

public:
    Treap() : root(nullptr) {}
    ~Treap() { clear(root); }

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

    void insert(int key) {
        static mt19937 gen(random_device{}());
        static uniform_int_distribution<int> dist(1, 1000000);
        root = insert(root, key, dist(gen));
    }

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

    int maxDepth() {
        if (!root) return 0;
        queue<pair<TreapNode*, int>> q;
        q.push({ root, 1 });
        int maxDepth = 0;

        while (!q.empty()) {
            auto current = q.front();
            q.pop();
            maxDepth = max(maxDepth, current.second);

            if (current.first->left)
                q.push({ current.first->left, current.second + 1 });
            if (current.first->right)
                q.push({ current.first->right, current.second + 1 });
        }
        return maxDepth;
    }

    vector<int> getAllDepths() {
        vector<int> depths;
        getDepths(root, 1, depths);
        return depths;
    }
}

```

```

};

vector<int> generateRandomNumbers(int n) {
    vector<int> numbers(n);
    iota(numbers.begin(), numbers.end(), 1);
    shuffle(numbers.begin(), numbers.end(), mt19937(random_device{}()));
    return numbers;
}

vector<int> generateOperationNumbers(int n, int count) {
    vector<int> numbers;
    mt19937 gen(random_device{}());
    uniform_int_distribution<int> dist(1, 2 * n);

    for (int i = 0; i < count; ++i) {
        numbers.push_back(dist(gen));
    }
    return numbers;
}

void runTests(int n, ofstream& outFile) {
    double avlInsertTime = 0, treapInsertTime = 0;
    double avlRemoveTime = 0, treapRemoveTime = 0;
    double avlSearchTime = 0, treapSearchTime = 0;
    int avlMaxDepth = 0, treapMaxDepth = 0;
    vector<vector<int>> allAvlDepths, allTreapDepths;

    for (int rep = 0; rep < 50; ++rep) {
        vector<int> numbers = generateRandomNumbers(n);

        AVLTree avlTree;
        Treap treap;

        for (int num : numbers) {
            avlTree.insert(num);
            treap.insert(num);
        }

        avlMaxDepth += avlTree.maxDepth();
        treapMaxDepth += treap.maxDepth();

        allAvlDepths.push_back(avlTree.getAllDepths());
        allTreapDepths.push_back(treap.getAllDepths());

        vector<int> insertNumbers = generateOperationNumbers(n, 1000);
        auto start = high_resolution_clock::now();
        for (int num : insertNumbers) {
            avlTree.insert(num);
        }
        auto end = high_resolution_clock::now();
        avlInsertTime += duration_cast<microseconds>(end - start).count() / 1000.0;

        start = high_resolution_clock::now();
        for (int num : insertNumbers) {
            treap.insert(num);
        }
        end = high_resolution_clock::now();
        treapInsertTime += duration_cast<microseconds>(end - start).count() / 1000.0;

        vector<int> removeNumbers = generateOperationNumbers(n, 1000);
        start = high_resolution_clock::now();
        for (int num : removeNumbers) {
            avlTree.remove(num);
        }
        end = high_resolution_clock::now();
        avlRemoveTime += duration_cast<microseconds>(end - start).count() / 1000.0;

        start = high_resolution_clock::now();

```

```

    for (int num : removeNumbers) {
        treap.remove(num);
    }
    end = high_resolution_clock::now();
    treapRemoveTime += duration_cast<microseconds>(end - start).count() / 1000.0;

    vector<int> searchNumbers = generateOperationNumbers(n, 1000);
    start = high_resolution_clock::now();
    for (int num : searchNumbers) {
        avlTree.search(num);
    }
    end = high_resolution_clock::now();
    avlSearchTime += duration_cast<microseconds>(end - start).count() / 1000.0;

    start = high_resolution_clock::now();
    for (int num : searchNumbers) {
        treap.search(num);
    }
    end = high_resolution_clock::now();
    treapSearchTime += duration_cast<microseconds>(end - start).count() / 1000.0;
}

// Усреднение результатов
avlMaxDepth /= 50;
treapMaxDepth /= 50;
avlInsertTime /= 50;
treapInsertTime /= 50;
avlRemoveTime /= 50;
treapRemoveTime /= 50;
avlSearchTime /= 50;
treapSearchTime /= 50;

// Усреднение глубин веток
vector<double> avgAvlDepths, avgTreapDepths;
if (!allAvlDepths.empty()) {
    avgAvlDepths.resize(allAvlDepths[0].size(), 0);
    for (const auto& depths : allAvlDepths) {
        for (size_t i = 0; i < min(avgAvlDepths.size(), depths.size()); ++i) {
            avgAvlDepths[i] += depths[i];
        }
    }
    for (auto& depth : avgAvlDepths) {
        depth /= allAvlDepths.size();
    }
}

if (!allTreapDepths.empty()) {
    avgTreapDepths.resize(allTreapDepths[0].size(), 0);
    for (const auto& depths : allTreapDepths) {
        for (size_t i = 0; i < min(avgTreapDepths.size(), depths.size()); ++i) {
            avgTreapDepths[i] += depths[i];
        }
    }
    for (auto& depth : avgTreapDepths) {
        depth /= allTreapDepths.size();
    }
}

// Запись результатов в CSV
outFile << n << ","
    << avlMaxDepth << "," << avlInsertTime << "," << avlRemoveTime << "," <<
avlSearchTime << ",";
for (double depth : avgAvlDepths) {
    outFile << depth << " ";
}
outFile << ",";

```

```

        outFile << treapMaxDepth << "," << treapInsertTime << "," << treapRemoveTime << ","
<< treapSearchTime << ",";
        for (double depth : avgTreapDepths) {
            outFile << depth << " ";
        }
        outFile << endl;
    }

int main() {
    ofstream outFile("tree_comparison_results.csv");

    if (!outFile) {
        cerr << "Failed to open output file." << endl;
        return 1;
    }

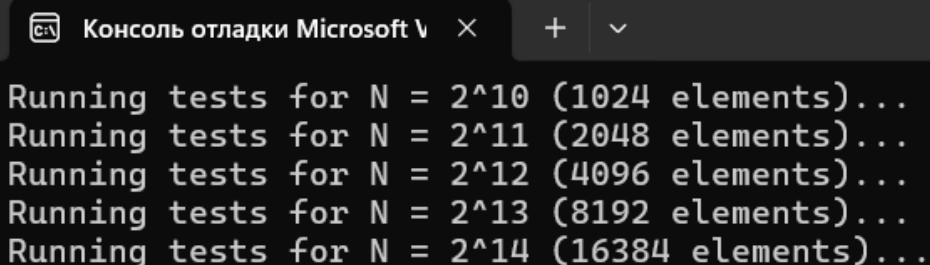
    // Заголовки CSV
    outFile << "N,"
        <<
        "AVL_MaxDepth,AVL_InsertTime(ms),AVL_RemoveTime(ms),AVL_SearchTime(ms),AVL_BranchDepths,"
        <<
        "Treap_MaxDepth,Treap_InsertTime(ms),Treap_RemoveTime(ms),Treap_SearchTime(ms),Treap_Bran
chDepths"
        << endl;

    for (int i = 10; i <= 18; ++i) {
        int n = pow(2, i);
        cout << "Running tests for N = 2^" << i << " (" << n << " elements)..." << endl;
        runTests(n, outFile);
    }

    outFile.close();
    cout << "All tests completed. Results saved to tree_comparison_results.csv" << endl;
    return 0;
}

```

Результат работы программы:



```

Консоль отладки Microsoft V
Running tests for N = 2^10 (1024 elements)...
Running tests for N = 2^11 (2048 elements)...
Running tests for N = 2^12 (4096 elements)...
Running tests for N = 2^13 (8192 elements)...
Running tests for N = 2^14 (16384 elements)...

```

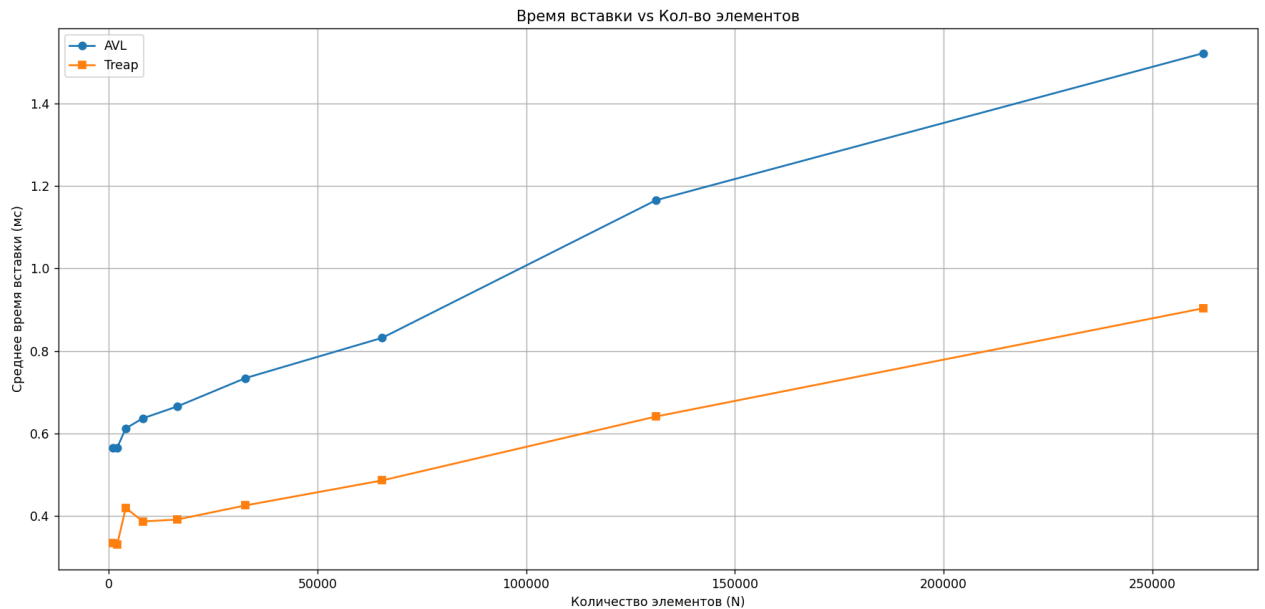
вывод результатов в файл

[illegible]

Графики:

язык программирования Python

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



- **Оси:**

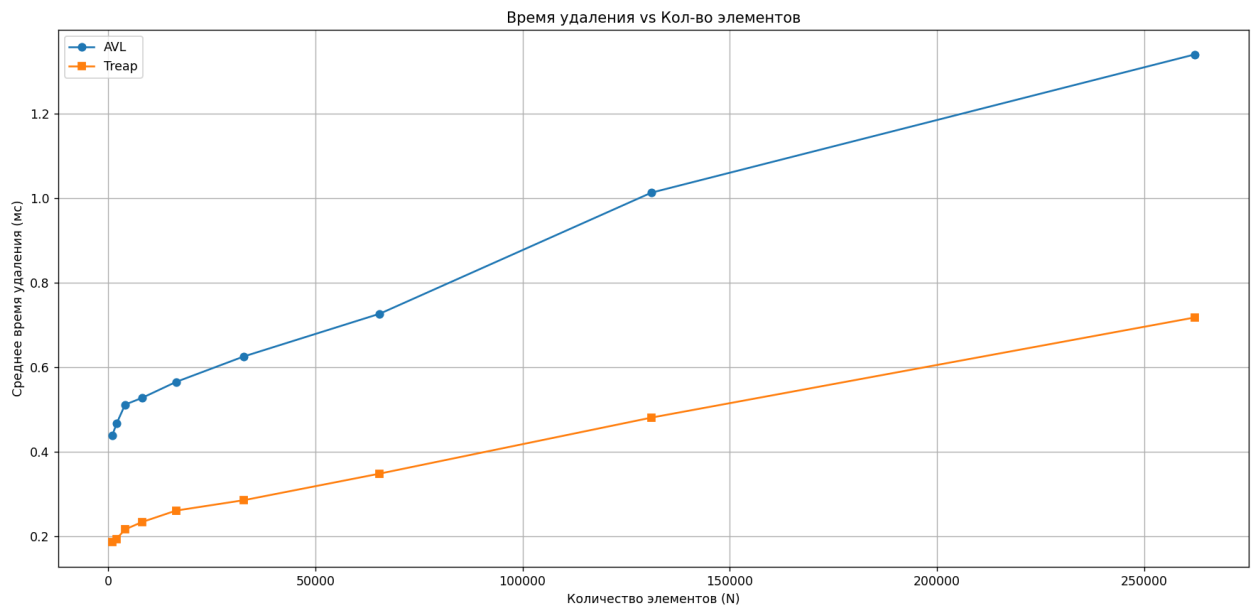
X — количество элементов (N),

Y — среднее время вставки (в мс)

- **Выводы:**

- ❖ У AVL из-за балансировки вставка может быть чуть медленнее, особенно при больших N.
- ❖ У Treap вставка может быть быстрее на случайных данных, потому что балансировка — рандомизированная.
- ❖ Но если данные отсортированы, у Treap может страдать производительность, если приоритеты не помогают сохранить баланс.

2. График зависимости среднего времени удаления от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.



- **Оси:**

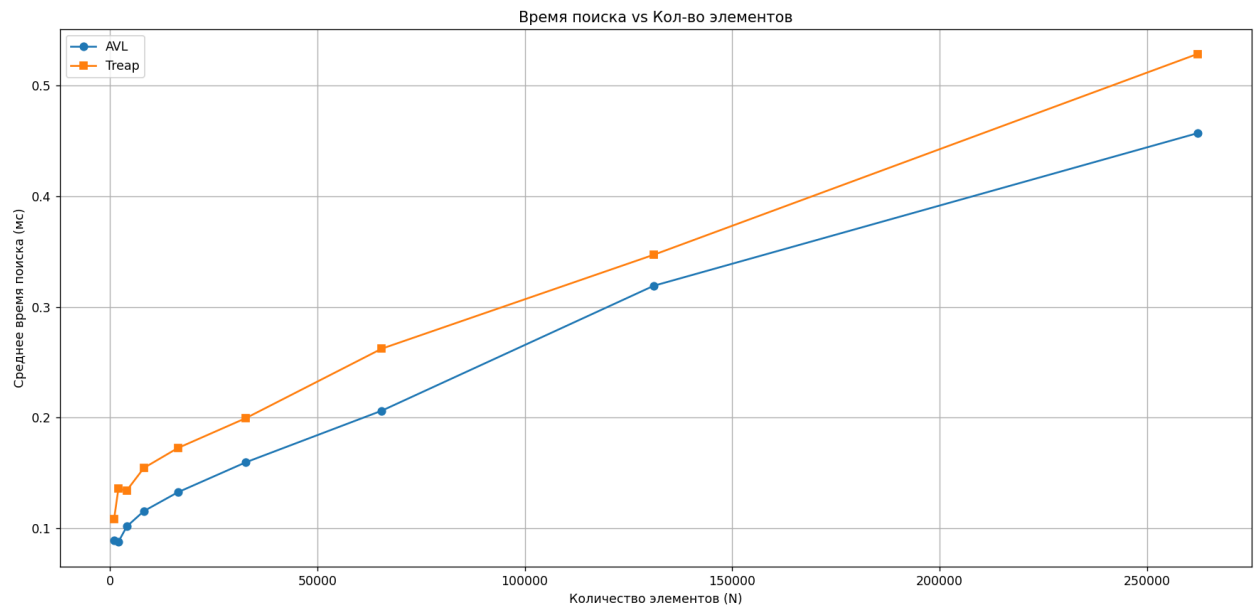
X — N,

Y — среднее время удаления

- **Выводы:**

- ❖ Удаление в AVL — относительно тяжёлый процесс, потому что нужно выполнять ребалансировку.
- ❖ В Треар удаление обычно быстрее, но всё зависит от структуры дерева (в частности — от случайных приоритетов).
- ❖ Если Треар работает на почти отсортированных данных, удаление может замедляться.

3. График зависимости среднего времени поиска от количества элементов в изначальном дереве для вашего варианта дерева и AVL дерева.



- **Оси:**

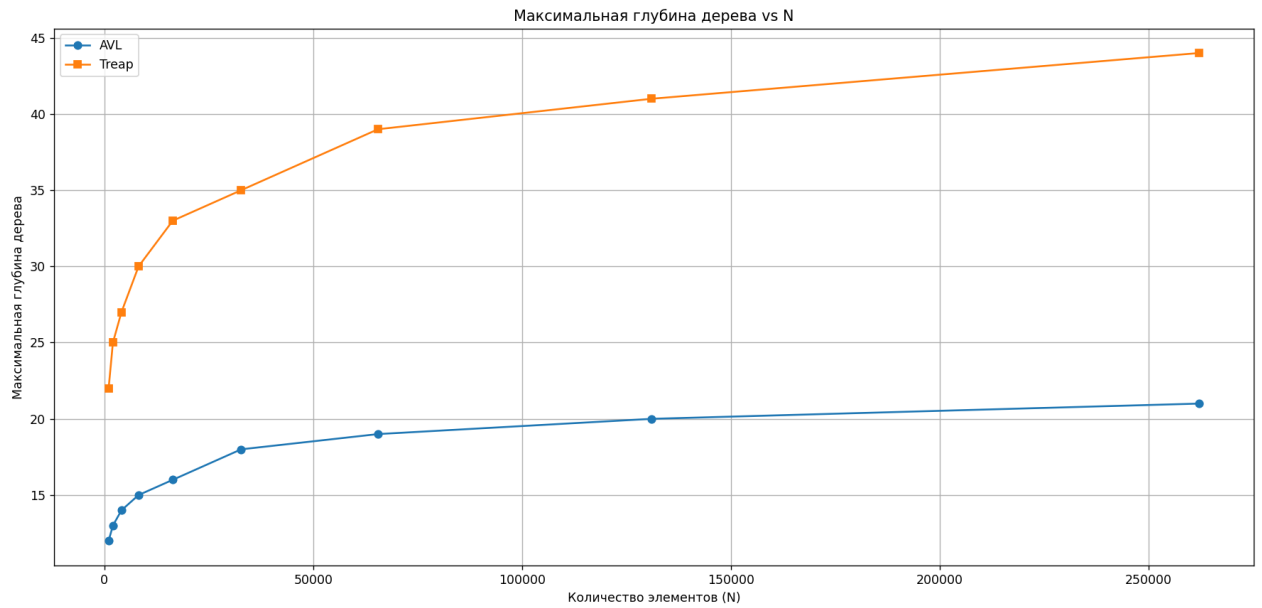
X — N,

Y — среднее время поиска

- **Выводы:**

- ❖ Поиск у AVL очень стабилен, так как дерево всегда сбалансировано ($O(\log N)$).
- ❖ У Treap поиск может быть медленнее на некоторых N, особенно если дерево стало разбалансированным.

4. График максимальной высоты полученного дерева в зависимости от N.



- **Оси:**

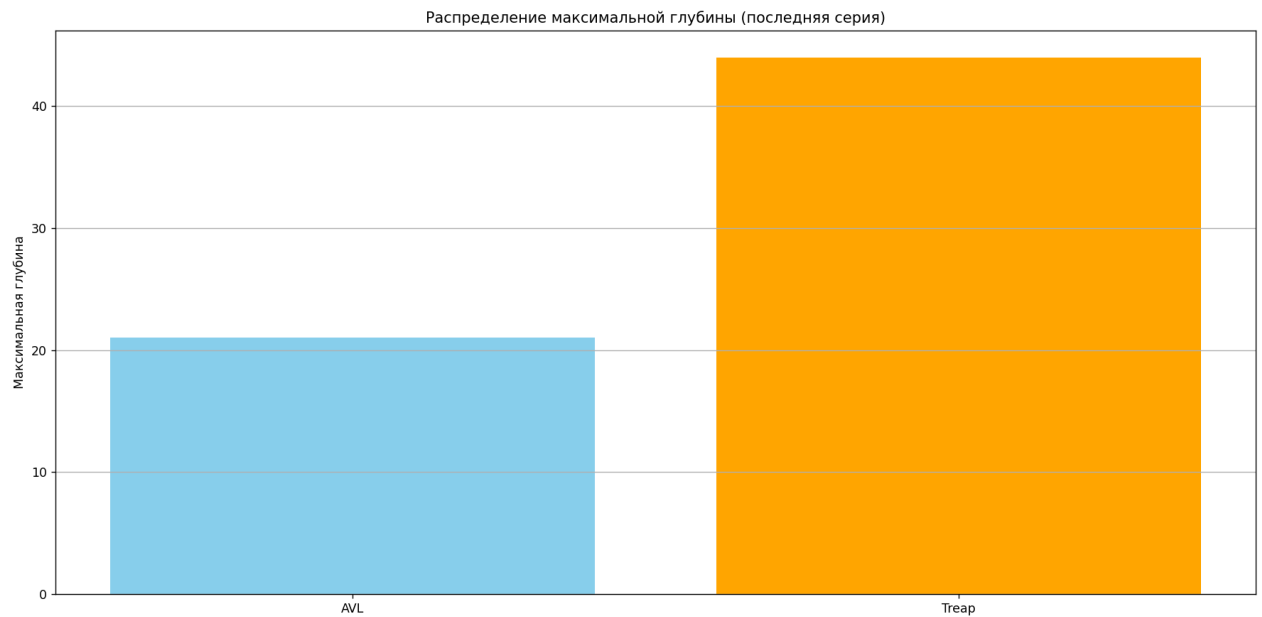
X — N,

Y — максимальная глубина дерева

- **Выводы:**

- ❖ У AVL рост должен быть плавным, около $\log_2 N$.
- ❖ У Treap возможны скачки, особенно на некоторых входах.
- ❖ Если Treap сильно выходит за $\log_2 N$, значит, дерево разбалансировалось.

5. Гистограмму среднего распределения максимальной высоты для последней серии тестов для AVL и для вашего варианта.



- **Оси:**

X — дерево,

Y — максимальная глубина в последней серии (одно значение для AVL, одно — для Tgear)

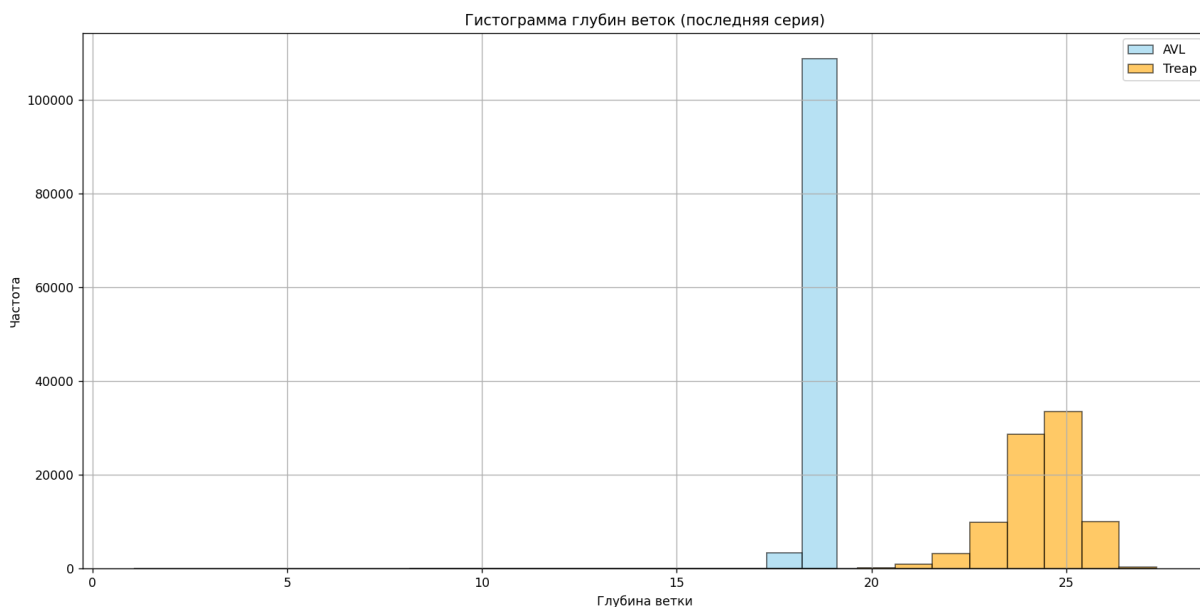
- **Что видно:**

Конкретная глубина дерева при последнем запуске.

- **Выводы:**

- ❖ Хороший способ быстро сравнить структуру деревьев.
- ❖ Если Tgear сильно выше — это потенциальная проблема разбалансировки.

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



- **Оси:**

X — дерево,

Y — средняя глубина веток

- **Что видно:**

Как в среднем выглядят пути от корня до листьев.

- **Выводы:**

- ❖ У AVL дерево сбалансировано — средняя глубина будет ближе к $\log_2 N$.
- ❖ У Treap может быть больше разброс — глубина варьируется, особенно если дерево не очень сбалансировано.
- ❖ Если у Treap средняя глубина заметно выше, значит, он далек от идеального баланса.

ВЫВОД

1. Сравнение производительности операций во вставке, удалении и поиске.

На основании построенных графиков видно, что:

- AVL-деревья демонстрируют стабильное и предсказуемое время выполнения всех операций благодаря строгой балансировке.
- Treap(декартово дерево) показывает сопоставимое или лучшее время вставки и удаления на случайных данных, за счёт рандомизированного подхода к балансировке.

Таким образом, в среднем производительность Treap может быть выше, но гарантии логарифмической сложности нет, в отличие от AVL.

2. Преимущества и особенности сбалансированных деревьев

AVL-дерево поддерживает баланс: после каждой модификации дерева проводится ребалансировка, что позволяет сохранять высоту порядка $O(\log N)$. Это обеспечивает:

- Стабильную скорость поиска, вставки и удаления.
- Надежную работу независимо от распределения входных данных.

Treap(декартово дерево), с другой стороны:

- Опирается на вероятностную балансировку — высота дерева в среднем $O(\log N)$, но в худшем случае может достигать $O(N)$.
- Это упрощает реализацию и ускоряет выполнение операций в среднем, но снижает предсказуемость поведения.

3. Анализ высоты дерева

По графику зависимости максимальной высоты от количества элементов:

- Высота AVL-дерева растёт логарифмически, что подтверждает корректную балансировку.
- Treap(декартово дерево) может иметь большую высоту. Это подтверждается и гистограммой максимальной высоты, где значения для Treap(декартово дерево) иногда выше.

Гистограмма средней глубины веток также показывает:

- У AVL-дерева средняя глубина ветвей находится близко к $\log N$.
- У Treap(декартово дерево) наблюдается более широкое распределение, что отражает потенциальное разбалансирование дерева.

Итог:

Работа позволила глубже понять, насколько важен выбор структуры данных для эффективности алгоритмов. Использование сбалансированных деревьев, таких как AVL или Treap(декартово дерево), является предпочтительным решением для динамических множеств, так как они обеспечивают стабильное и предсказуемое логарифмическое время выполнения операций, независимо от порядка вставки данных. Treap благодаря своей случайной балансировке, сочетает преимущества бинарного поиска и вероятностного баланса, обеспечивая хорошую производительность без необходимости поддержания строгой балансировки, как в AVL-деревьях.

