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-38.....(Нергарян Геворг Гарегинович)
Ссылка на репозиторий: .....(https://github.com/MUCTR-IKT-CPP/Nergaryan_KC-38_Algos)

Приняли: .....Пысин Максим Дмитриевич
.....Краснов Дмитрий Олегович
.....Лобанов Алексей Владимирович
.....Крашенинников Роман Сергеевич

Дата сдачи: ……..23.05.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить декартово дерево.

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.

Декартово дерево, также известное как дерамид, является структурой данных, комбинирующей свойства двух других деревьев: бинарного дерева поиска (BST) и кучи. Оно представляет собой бинарное дерево, в котором каждый узел содержит два значения: ключ и приоритет. Ключи узлов удовлетворяют свойству BST, то есть ключ в левом поддереве меньше, чем ключ в узле, и ключ в правом поддереве больше. Приоритеты узлов определяют порядок обработки элементов и соответствуют свойству кучи, где приоритет родительского узла всегда больше приоритета его потомков.

Декартово дерево обеспечивает эффективное выполнение операций вставки, удаления и поиска элементов. При вставке нового элемента с ключом K и приоритетом P, алгоритм сначала находит место для вставки K в BST с учетом свойства упорядоченности ключей. Затем он рекурсивно спускается по дереву, сравнивая приоритет текущего узла с P и, если P меньше, выполняет повороты для поддержания свойства кучи. Таким образом, приоритеты узлов определяют балансировку дерева, а ключи упорядочивают элементы.

Одним из основных преимуществ декартового дерева является его гибкость и возможность использования для различных задач. Например, оно может быть использовано для выполнения операций с максимальным/минимальным значением, реализации очереди с приоритетом или сортировки элементов.

Важно отметить, что декартово дерево не является самобалансирующимся. В отличие от AVL-дерева, которое поддерживает строгое балансирование высоты поддеревьев, декартово дерево не обязательно соблюдает балансировку по высоте. Вместо этого оно обеспечивает вероятностную балансировку с использованием приоритетов. Это может привести к неравномерному распределению высоты поддеревьев и потенциально худшей временной сложности операций.

Однако декартово дерево имеет другие преимущества, такие как простота реализации, отсутствие необходимости хранить дополнительные поля для балансировки и поддержку операций вставки и удаления в среднем случае с амортизированной сложностью O(log N), где N - количество элементов в дереве.

В заключение, декартово дерево представляет собой эффективную и гибкую структуру данных, сочетающую свойства бинарного дерева поиска и кучи. Оно обеспечивает эффективные операции вставки, удаления и поиска элементов, но не гарантирует строгую балансировку высоты поддеревьев, как в случае AVL-дерева. Декартово дерево может быть полезным для решения различных задач, требующих комбинации упорядоченности и приоритета.

Выполнение задачи.

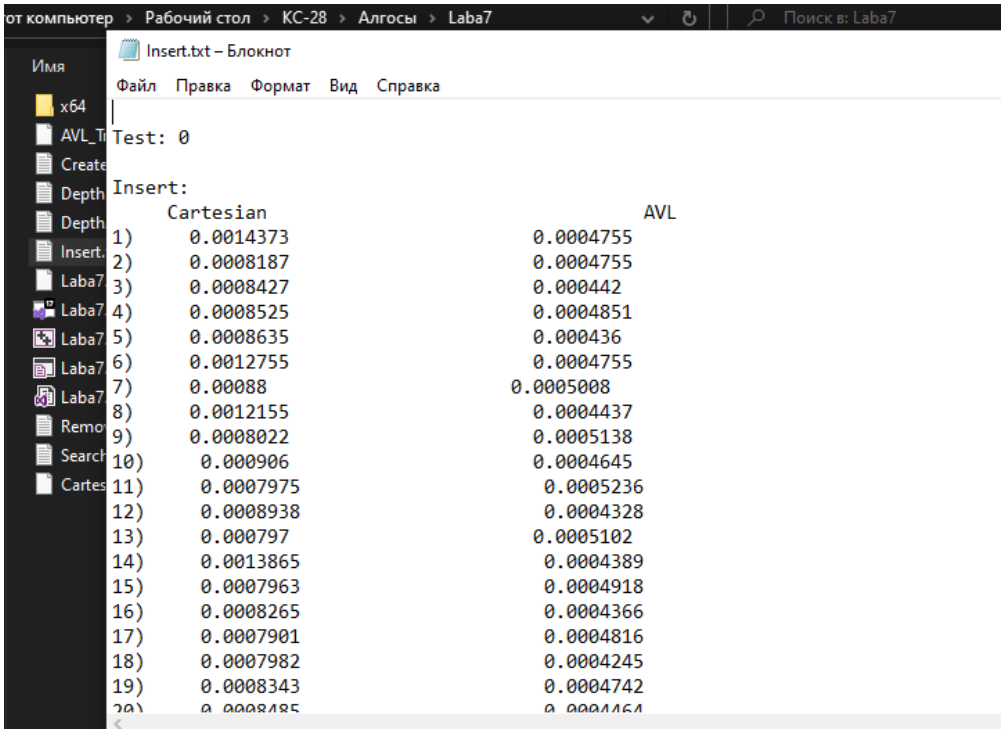
Код на Языке C++

```
#include<iostream>
#include <climits>
#include <fstream>
#include <chrono>
#include"Cartesian_tree.h"
#include"AVL_Tree.h"
#include<vector>

using namespace std;

//Генерируем массив
void generate(int* arr, int N, bool random) {

    cout << "Post 1.вектор" << endl;
    for (int i = 0; i < N; i++) {
        if (random) {
            int ind = 0 + rand() % (N - 1);
            arr[i] = ind;
        }
        else
            arr[i] = i;
    }
}

//Вставка массива в деревья
void createTree(Cartesian_tree myTree, AVL_Tree myAVL, int N, int* arr, int n, int i) {
    ofstream out1("Create.txt", ios::app);

    for (int i = 0; i < N; i++) {
        myTree.insert(arr[i]);
        myAVL.insert(arr[i]);
    }
    if (out1.is_open())
    {
        out1 << n + 1 << ")     " << myTree.find_Depth() << "                        " <<
myAVL.find_Depth() << endl;
    }
    cout << N << endl;
    if ((i == 7) && (n == 49)) {
        myTree.printAllDepth(n);
        myAVL.printAllDepth(n);
    }
}

//теперь вставка элемента в деревья
void insertInTree(Cartesian_tree myTree, AVL_Tree myAVL, int N, int n) {
    int num;
    ofstream out("Insert.txt", ios::app);
    chrono::high_resolution_clock::time_point start1 = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        num = N + rand() % (N + 1000);
        myTree.insert(num);
    }
    chrono::high_resolution_clock::time_point end1 = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff1 = end1 - start1;

    chrono::high_resolution_clock::time_point start2 = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        num = N + rand() % (N + 1000);
        myAVL.insert(num);
    }
    chrono::high_resolution_clock::time_point end2 = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff2 = end2 - start2;
```

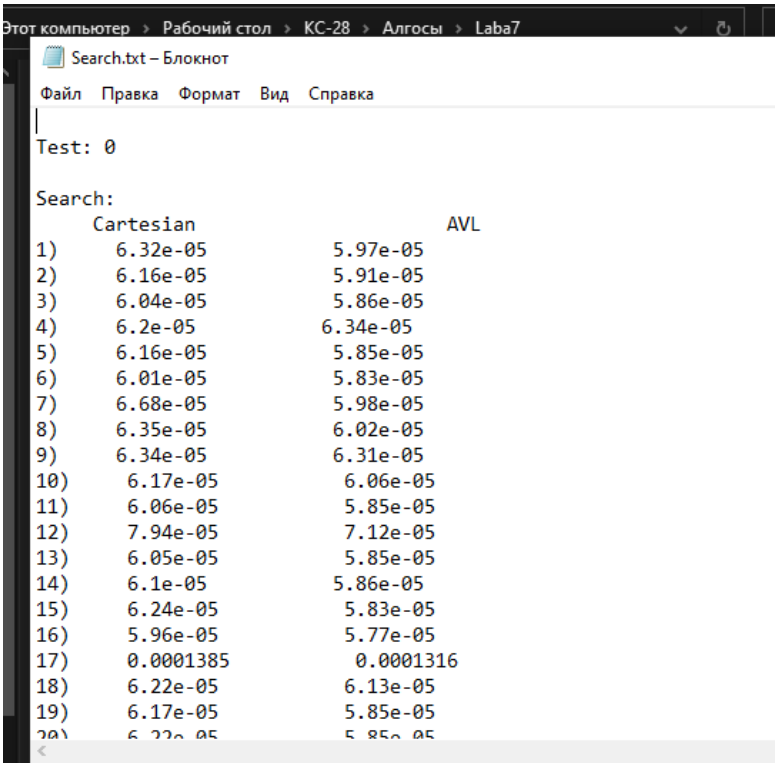
```
    if (out.is_open())
    {
        out << n + 1 << ")     " << sec_diff1.count() << "                        " <<
sec_diff2.count() << endl;
    }
}


//Поиск элемента
void findElements(Cartesian_tree myTree, AVL_Tree myAVL, int N, int n) {
    int num;
    ofstream out2("Search.txt", ios::app);
    chrono::high_resolution_clock::time_point start1 = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        num = 0 + rand() % (N - 1);
        myTree.search(num);
    }
    chrono::high_resolution_clock::time_point end1 = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff1 = end1 - start1;

    chrono::high_resolution_clock::time_point start2 = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        num = 0 + rand() % (N - 1);
        myAVL.search(num);

    }
    chrono::high_resolution_clock::time_point end2 = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff2 = end2 - start2;

    if (out2.is_open())
    {
        out2 << n + 1 << ")     " << sec_diff1.count() << "           " << sec_diff2.count()
<< endl;
    }
}

//Удаение элемента
void removeTree(Cartesian_tree myTree, AVL_Tree myAVL, int* arr, int N, int n) {

    ofstream out3("Remove.txt", ios::app);
    chrono::high_resolution_clock::time_point start1 = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        myTree.remove(arr[N - i]);
    }
    chrono::high_resolution_clock::time_point end1 = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff1 = end1 - start1;

    chrono::high_resolution_clock::time_point start2 = chrono::high_resolution_clock::now();
    for (int i = 0; i < 1000; i++) {
        myAVL.remove(arr[N - i]);
    }
    chrono::high_resolution_clock::time_point end2 = chrono::high_resolution_clock::now();
    chrono::duration<double> sec_diff2 = end2 - start2;
    if (out3.is_open()) //  открываем файл для записи
    {
        out3 << n + 1 << ")      " << sec_diff1.count() << "              " <<
sec_diff2.count() << endl;
    }
}

//Тесты
void tests(int test, bool random) {
    //открываем поток записи в наш файл (с возможностью дописать)
    ofstream out1("Create.txt", ios::app);
    ofstream out("Insert.txt", ios::app);
    ofstream out2("Search.txt", ios::app);
    ofstream out3("Remove.txt", ios::app);
    ofstream out4("Depth.txt", ios::app);
    ofstream out5("DepthAVL.txt", ios::app);
```

```
    for (int i = 0; i < test; i++) {
        if (out.is_open() && out2.is_open() && out3.is_open() && out1.is_open() &&
out4.is_open() && out5.is_open()) //открываем файл для записи
        {
            out << "\n" << "Test: " << i << endl;
            out1 << "\n" << "Test: " << i << endl;
            out2 << "\n" << "Test: " << i << endl;
            out3 << "\n" << "Test: " << i << endl;
            out4 << "\n" << "Test: " << i << endl;
            out5 << "\n" << "Test: " << i << endl;
        }
        if (out1.is_open())
        {
            out1 << "\nDepth: \n     Cartesian                                    AVL" << endl;
        }
        if (out.is_open())
        {
            out << "\nInsert: \n     Cartesian                                    AVL" << endl;
        }
        if (out2.is_open())
        {
            out2 << "\nSearch: \n     Cartesian                      AVL" << endl;
        }
        if (out3.is_open())
        {
            out3 << "\n Remove : \n     Cartesian                   AVL " << endl;
        }
        if (out4.is_open())
        {
            out4 << "\nDepth: \n  " << endl;
        }
        if (out5.is_open())
        {
            out5 << "\nDepthAVL: \n " << endl;
        }
        int N = pow(2, 10 + i);
        for (int n = 0; n < 50; n++) {
            cout << "Post 1" << endl;
            int* arr = new int[N];
            Cartesian_tree myTree;
            AVL_Tree myAVL;
            cout << "Post 1.6" << endl;
            generate(arr, N, random);
            cout << "Post 2.0" << endl;
            createTree(myTree, myAVL, N, arr, n, i);
            cout << "Post 2" << endl;
            insertInTree(myTree, myAVL, N, n);
            cout << "Post 3" << endl;
            findElements(myTree, myAVL, N, n);
            cout << "Post 4" << endl;
            removeTree(myTree, myAVL, arr, N, n);
            cout << "Post 5" << endl;

            delete(arr);
        }

    }
}
int main()
{

    srand(time(NULL));
    setlocale(LC_ALL, "Russian");
    tests(8, true);

    cout << "Успех" << endl;
      }
```

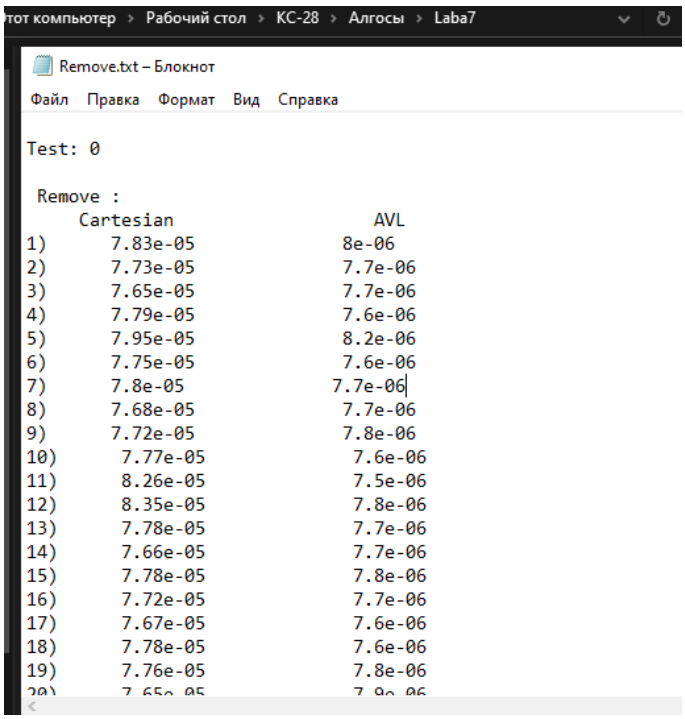
1000 вставок и их время



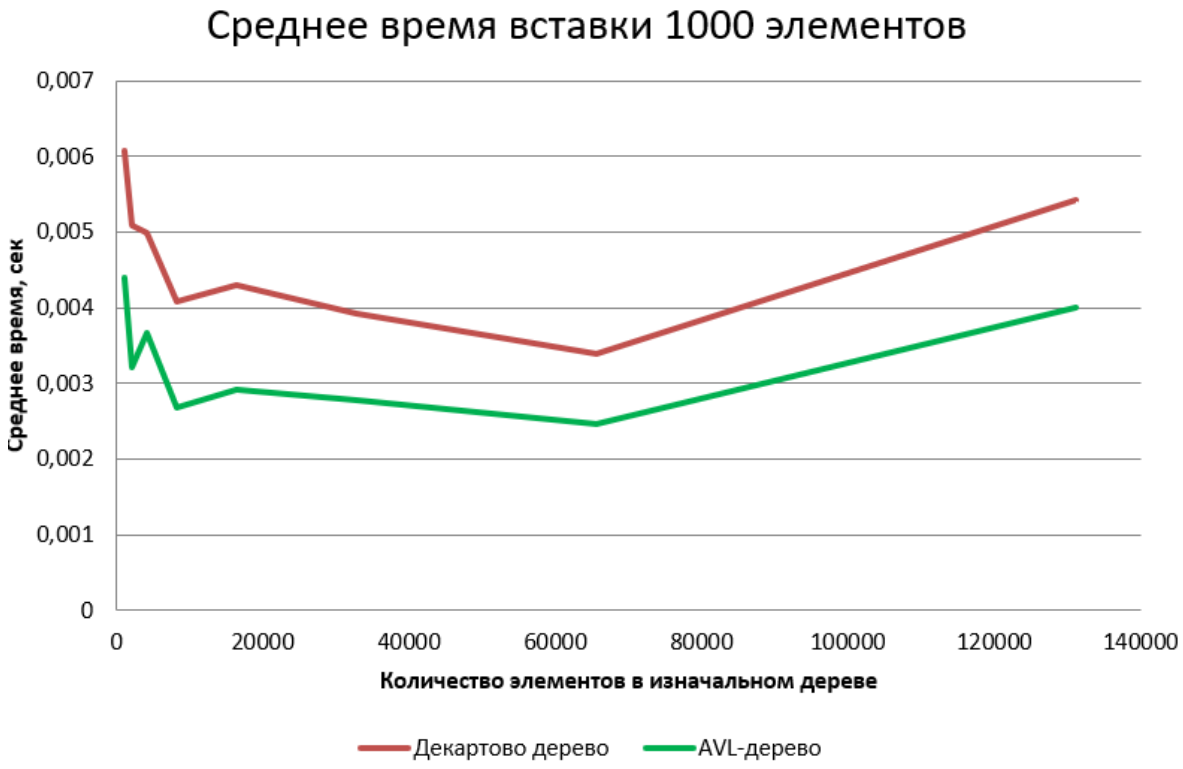
1000 поисков и время

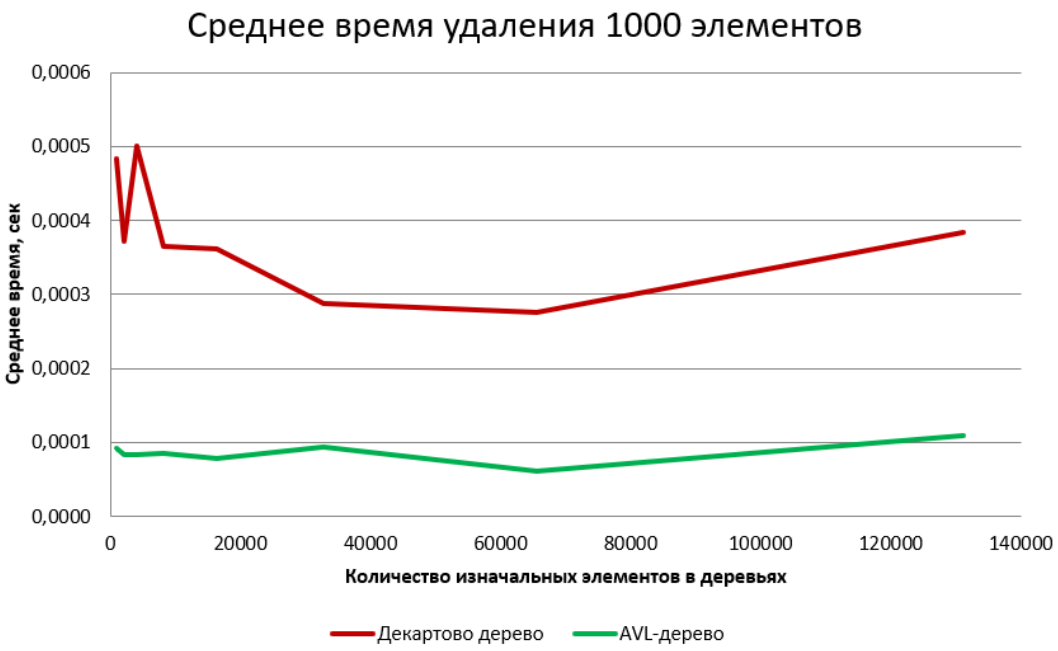


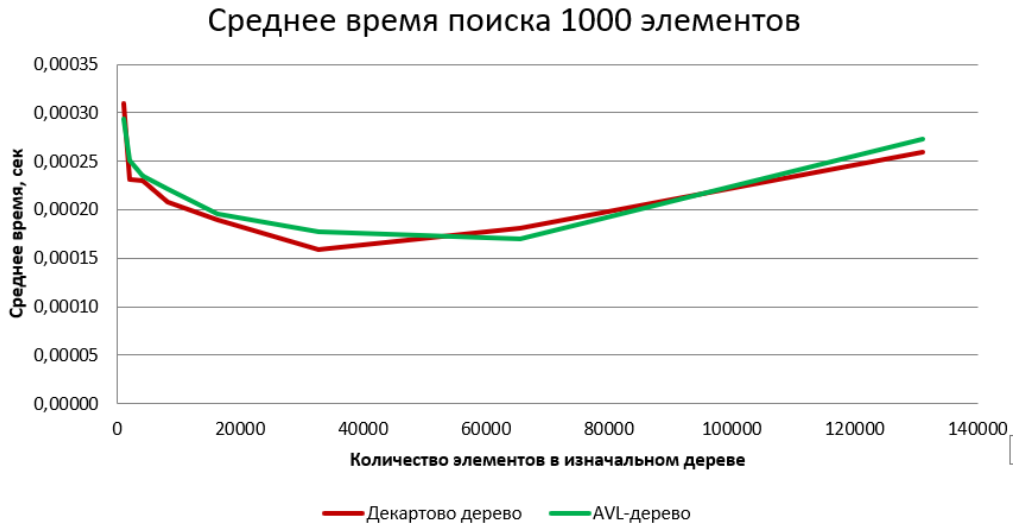
1000 удалений и время

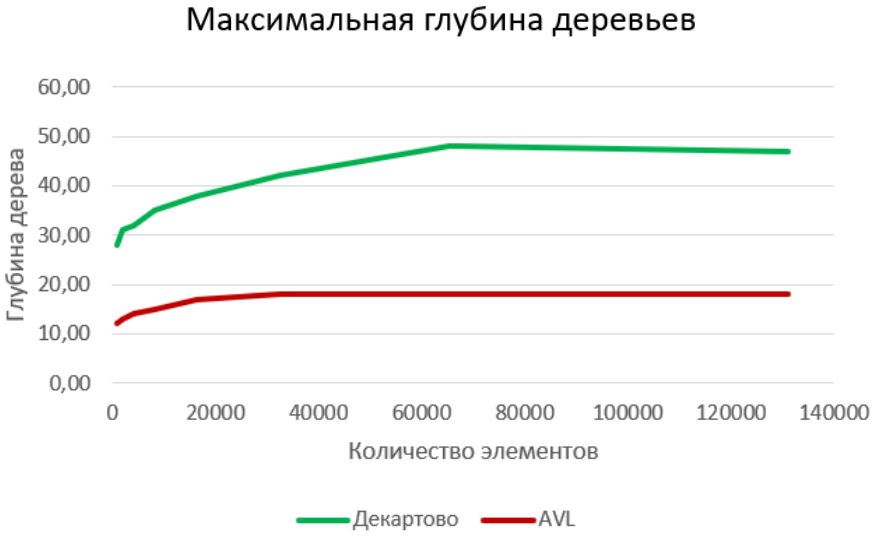


Прописываем функцию тестов, в которой производим 8 тестов по 50 циклов генераций и операций, указанных выше.

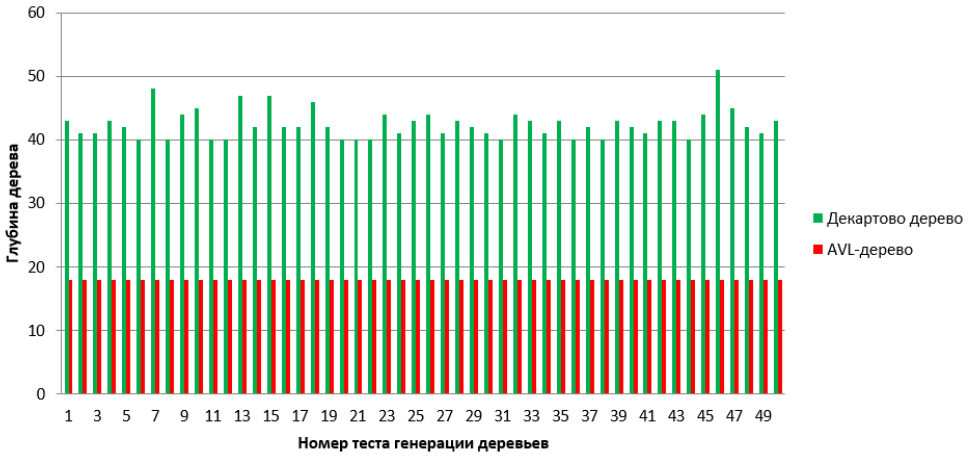



Среднее время удаления 1000 элементов
0,0006
0,0005
0,0004
0,0003
0,0002
0,0001
0,0000
Среднее время, сек
0
20000
40000
60000
80000
100000
120000
140000
Количество изначальных элементов в деревьях
Декартово дерево
AVL-дерево

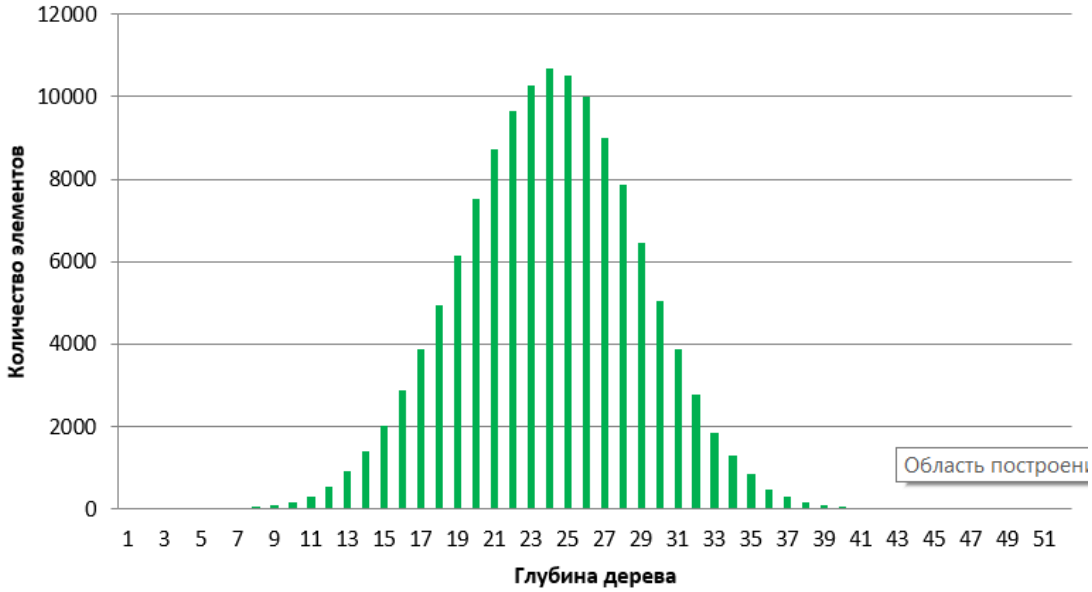

Среднее время поиска 1000 элементов
0,00035
0,00030
0,00025
0,00020
0,00015
0,00010
0,00005
0,00000
Среднее время, сек
0
20000
40000
60000
80000
100000
120000
140000
Количество элементов в изначальном дереве
Декартово дерево
AVL-дерево


Максимальная глубина деревьев
60,00
50,00
40,00
30,00
20,00
10,00
0,00
Глубина дерева
0
20000
40000
60000
80000
100000
120000
140000
Количество элементов
Декартово
AVL

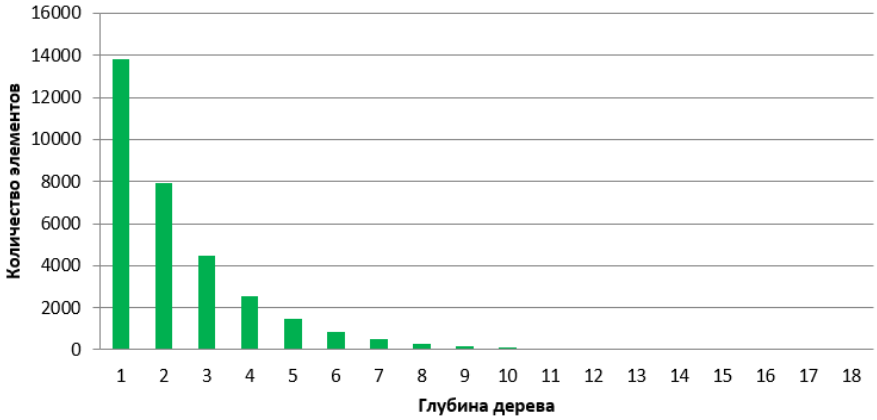
Среднее распределение максимальной глубины



Среднее распределение высот веток Декартова дерева



Среднее распределение высот веток AVL-дерева



Заключение.

Исходя из выполненной работы, мы можем сделать следующие выводы:

Реализация декартового дерева поиска оказалась проще по сравнению с AVL-деревом. Декартово дерево имеет более простую структуру и не требует дополнительных полей для балансировки, что делает его более доступным для реализации и понимания.

В отношении операции поиска, AVL-дерево и декартово дерево имеют схожую производительность. В некоторых случаях декартово дерево может показывать более высокую эффективность поиска, но в целом оба дерева выполняют эту операцию с приемлемой скоростью.

Однако, при заполнении и удалении элементов, декартово дерево показывает более длительное время выполнения по сравнению с AVL-деревом. Заполнение и удаление элементов в AVL-дереве более оптимизированы, что позволяет этому дереву быть более эффективным в таких операциях.

С учетом вышесказанного, для большинства случаев более предпочтительным выбором является AVL-дерево. Оно обеспечивает сравнимую скорость поиска с декартовым деревом, но при этом имеет более быстрые операции заполнения и удаления элементов.

Таким образом, при работе с деревьями поиска, рекомендуется использовать AVL-дерево вместо декартового дерева. AVL-дерево обеспечивает хороший баланс между скоростью поиска и операциями заполнения/удаления, что делает его эффективным выбором для большинства задач.