Министерство науки и высшего образования Российской Федерации
Российский химико-технологический университет имени Д. И. Менделеева

Факультет информационных технологий и управления
Кафедра информационных компьютерных технологий

ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ №7

«Декартово дерево»

Выполнил студент группы КС-30 Колесников Артем Максимович

Ссылка на репозиторий: https://github.com/MUCTR-IKT-CPP/AMKolesnikov_30_ALG

Приняли: аспирант кафедры ИКТ Пысин Максим Дмитриевич
аспирант кафедры ИКТ Краснов Дмитрий Олегович

Дата сдачи: 25.04.2022

Москва
2022

Содержание

Описание задачи

В рамках лабораторной работы необходимо изучить и реализовать декартово дерево поиска. 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 структуры

Бинарное (двоичное) дерево поиска – это бинарное дерево, для которого выполняются следующие дополнительные условия (свойства дерева поиска):

- оба поддерева – левое и правое, являются двоичными деревьями поиска;
- у всех узлов левого поддерева произвольного узла X значения ключей данных меньше, чем значение ключа данных самого узла X;
- у всех узлов правого поддерева произвольного узла X значения ключей данных не меньше, чем значение ключа данных узла X.

Деревья — это замечательная структура, и в среднем работает очень быстро. Однако, важно помнить, что асимптотическая сложность каждой из вышеописанных операций равна, по сути, O(h), где h это высота нашего дерева. И учитывая эту особенность, мы понимаем, что в худшем случае, когда дерево будет строиться оно может построиться так, что высота итогового дерева будет равна количеству добавляемых в него элементов, что как раз и составляет худший вариант работы с деревом. А когда это может произойти? Когда массив

данных, на основании которых создается дерево, был отсортирован. Решается эта проблема при помощи создание таких структур, которые на основании своих правил работы в итоге всегда получаются сбалансированными.

АВЛ дерево является обычным двоичным деревом поиска, следовательно его правое поддерево всегда меньше значения корня, а правое поддерево всегда больше. При это, при построении дерева мы руководствуемся правилом балансировки или перебалансировки: для любого узла дерева высота его правого поддерева отличается от высоты левого поддерева не более чем на единицу. Является доказанным, что при соблюдении этого правила высота дерева логарифмически зависит от количества элементов, добавляемых в дерево, т.е. h = O(log(n)).

Операция вставки и удаления вызываются практически так же, как у обычного двоичного дерева. Разница только в том, что для каждой вставки и каждого удаления требуется последним действием вызывать операцию перебалансировки, каждый раз проверяя от низа к верху необходимость ребалансировать дерево в текущем узле.
Операция балансировки вызывается тогда, когда фактор балансировки становится равным 2 или -2, т.е. тогда, когда разница между правым и левым поддеревьями является больше чем заложено в правиле. В этом случае требуется выполнить операцию поворота дерева, которая переориентирует узлы так, что они изменяют свое положение решая проблему дисбаланса.

Выделяют 4 основных поворота для AVL дерева:

- Простой/малый левый поворот
- Простой/малый правый поворот
- Сложный/большой левый поворот – это два последовательных поворота, сначала правый потом левый.
- Сложный/большой правый поворот – это два последовательных поворота, сначала левый потом правый.

Все повороты выполняются относительно какого-либо узла.

Декартово дерево, это двоичное дерево поиска, которое является достаточно популярной и простой реализацией самобалансирующегося варианта дерева. Декартово дерево в каждом узле помимо ключа, хранит так же приоритет узла, который отражает позицию элемента в такой структуре данных как куча. О куче мы будем подробно говорить в следующих лекциях, пока же, укажем, что куча — это древовидная структура, у которой родитель дерева больше всех его потомков (или меньше). По этой причине декартово дерево часто называют treap = tree + heap. Такое дерево называется декартовым, по той причине, что его узлы можно уложить на координатной плоскости, где x это ключ, а y это приоритет.

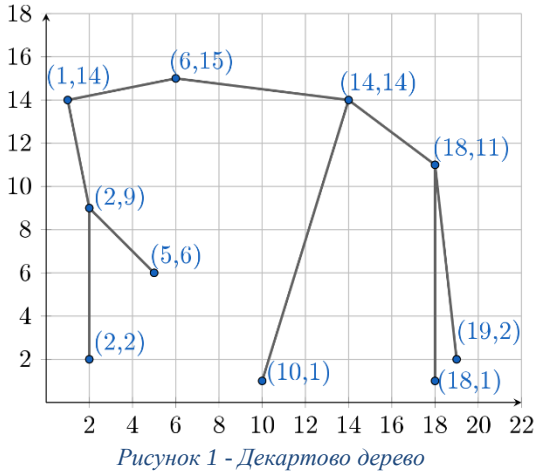

Рисунок 1 - Декартово дерево

Для построения декартового дерева нам потребуется:

1. Множество ключей, это те значения, которые есть в наших исходных данных, по которым мы и хотим построить декартово дерево поиска.
2. 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 иметь случайную природу, например можно использовать дату рождения.

Если мы делаем все правильно, то в итоге из множества деревьев которые мы можем построить используя только пункт 1, мы гарантированно построим только один единственный вариант существующий для пункта 1 + 2. Высота такого дерева с высокой вероятностью будет не превышать 4 * log (N), а значит будет логарифмической.
Стоит упомянуть два особенных момента:

- Одинаковые ключи следует хранить либо только в правом поддереве, либо только в левом, и тогда они не будут доставлять проблем.
- Одинаковых приоритетов стоит избегать, в идеале генерировать случайные числа от 0 до 1, но, если нужно, можно и случайное целочисленное число.

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.

Используя эти операции, мы можем выразить любую из других операций с декартовым деревом.

Выполнение задачи

Я реализовал декартово дерево поиска и avl дерево на языке C++. Моя программа состоит из 2 классов: «Treap», «AVL» и функции «main».

Класс «Treap» ~ занимается созданием декартовоа дерева поиска. Дерево строится на основе структуры «node», которая содержит ключ «x», приоритет «y» и ссылки на левого/правого потомка. В классе есть методы добавления элемента «insert», удаления элемента «remove», поиска элемента «search», вывода дерева в консоль «display» и расчета максимальной глубины «max_deep». Также в классе содержатся вспомогательные методы, которые осуществляют объединение двух деревьев «merge» и разделение дерева «split».

```
class Treap
{
private:

    // Определение узла декартового дерева
    struct Node
    {
        int x; // Хранимый ключ
        int y; // "Порядок" вершины
        int sz; // Размер поддерева (количество узлов, находящихся ниже данного)
        Node* l, * r; // Указатели на потомков данного узла

        // Конструктор узла (объявлен в описании самой структуры для простоты)
        Node(int _x)
        {
            x = _x;
            y = rand();
            sz = 1;
            l = r = nullptr;
        }
    };

    Node* root;


    void makeEmpty(Node* t)
    {
        if (t == nullptr)
            return;
        makeEmpty(t->l);
        makeEmpty(t->r);
        delete t;
    }

    // --- Основные операции - слияние (merge) и разделение (split) деревьев
    // Важно! При выполнении этих операций исходные деревья могут перестать быть
    // валидными!

    // Слияние двух деревьев (t1, t2). Работает корректно тогда и только тогда,
    // когда max(t1) < min(t2). Результат - корень нового дерева.
    Node* merge(Node* t1, Node* t2)
    {
        // Пустое дерево - то, у которого корень - nullptr.
        // Считаем, что слияние дерева с пустым деревом - это исходное дерево.
        if (t1 == nullptr) { return t2; }
        if (t2 == nullptr) { return t1; }
```

```
    // Дерево с большим значением y становится новым корнем
    if (t1->y > t2->y)
    {
        // Сливаем правое поддерево первого дерева со вторым деревом и ставим
        // результат в правое поддерево первого дерева.
        t1->r = merge(t1->r, t2);
        // Поскольку теперь дерево t1 поменялось, запускаем в нём обновление.
        update(t1);
        // Новый корень дерева - t1.
        return t1;
    }
    else
    {
        // Сливаем всё первое дерево с левым поддеревом второго дерева
        // (обязательно именно в таком порядке!!) и ставим результат в левое поддерево
        // второго дерева.
        t2->l = merge(t1, t2->l);
        // Пересчитываем значение в новом корне дерева
        update(t2);
        return t2;
    }
}

// Разделение дерева на два по заданному ключу. После этого в первом дереве
// будут значения (-inf, x), во втором - [x, +inf)
void split(Node* t, int x, Node*& t1, Node*& t2)
{
    // Пустое дерево будет в любом случае разделено на два пустых поддерева
    if (t == nullptr)
    {
        t1 = t2 = nullptr;
        return;
    }
    // Если текущая вершина содержит значение, меньшее, чем заданное,
    // то она отправляется в первое дерево. Правое поддерево данной вершины
    // в принципе может оказаться больше или равно x, так что его тоже разрезаем,
    // и записываем первое дерево-результат на его место.
    if (t->x < x)
    {
        split(t->r, x, t->r, t2);
        t1 = t;
    }
    else
    {
        // В противном случае "режем" левое поддерево. Рассуждения остаются такими
        // же (текущая вершина отправляется во второе дерево-результат)
        split(t->l, x, t1, t->l);
        t2 = t;
    }
    // В процессе отрезания мы модифицируем дерево, поэтому для данной вершины надо
    // пересчитать хранимое выражение
    update(t);
}

// Безопасное получение размера поддерева
int get_sz(Node* t)
{
    // Размер пустых деревьев считаем равным нулю
    if (t == nullptr) { return 0; }
    // У непустых поддеревьев размер хранится в корне
    return t->sz;
}

// Обновление размера поддерева
void update(Node* t)
```

```
{
    // Размер обновляем только для непустых деревьев
    if (t != nullptr)
    {
        t->sz = 1 + get_sz(t->l) + get_sz(t->r);
    }
}


// --- Операции, проводимые с деревом

// Добавление нового элемента x в дерево t. Корень дерева может измениться!
Node* add(Node*& t, int x)
{
    Node* t1, * t2;
    // Для добавления делаем следующее:
    // - Разрезаем исходное дерево по ключу x. В левом поддереве все элементы меньше x,
    //   в правом - не меньше.
    split(t, x, t1, t2);
    // - Создаём новое дерево из одной вершины - собственно, x.
    Node* new_tree = new Node(x);
    // - Производим слияние левого поддерева с новым, потом слияние результата с правым
    //   Результат слияния - новый корень дерева.
    t = merge(merge(t1, new_tree), t2);
    return t;
}

// Удаление вершины из дерева. В данном случае работать будет только с целыми числами!
Node* remove(Node*& t, int x)
{
    Node* t1, * t2, * t3, * t4;
    // Для удаления делаем следующее:
    // - Разрезаем исходное дерево по ключу x.
    split(t, x, t1, t2);
    // - Разрезаем правое поддерево по ключу x + 1 (вот зачем нужны были целые числа!)
    split(t2, x + 1, t3, t4);
    // - Соединяем деревья t1 и t4, которые теперь не содержат ключа x
    //   (он остался в дереве t3)
    t = merge(t1, t4);
    // - Очищаем память, занимаемую деревом t3 (если не хотим утечек)
    delete t3;
    return t;
}

// Вывод элементов дерева на экран в отсортированном порядке (обход ЛКП)
void print(Node* t)
{
    // У пустых деревьев выводить нечего
    if (t != nullptr)
    {
        // Сначала выводим всё из левого поддерева, затем то, что хранится в
        // корне, затем всё из правого поддерева
        print(t->l);
        cout << t->x << " ";
        print(t->r);
    }
}

void print_deep(Node* t)
{
    // У пустых деревьев выводить нечего
    if (t != nullptr)
    {
        // Сначала выводим всё из левого поддерева, затем то, что хранится в
        // корне, затем всё из правого поддерева
```

```
            print_deep(t->l);
            cout << t->sz << " ";
            print_deep(t->r);
        }
    }

    // Получение элемента, стоящего на k-м месте в дереве (начиная с единицы!)
    int get_k(Node* t, int k)
    {
        if (k < get_sz(t->l))
        {
            return get_k(t->l, k);
        }
        else if (k == get_sz(t->l))
        {
            return t->x;
        }
        else
        {
            return get_k(t->r, k - get_sz(t->l) - 1);
        }
    }

    Node* find(Node* t, int _x) {
        if (t == nullptr)
            return nullptr;
        else if (_x < t->x)
            return find(t->l, _x);
        else if (_x > t->x)
            return find(t->r, _x);
        else
            return t;
    }

    int depth(Node* t)
    {
        int ret = 0;
        if (t)
        {
            int lDepth = depth(t->l);
            int rDepth = depth(t->r);
            ret = std::max(lDepth + 1, rDepth + 1);
        }
        return ret;
    }

public:
    Treap() {
        root = nullptr;
    };
    ~Treap() {
        makeEmpty(root);
    };

    void insert(int x)
    {
        root = add(root, x);
    }

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

    void display()
```

```
    {
        print(root);
        cout << endl;
    }

    void display_deep()
    {
        print_deep(root);
        cout << endl;
    }

    void search(int x) {
        find(root, x);
    }

    int max_deep() {
        return depth(root);
    }
};
```

Класс «AVL» ~ занимается созданием avl дерева поиска. Дерево строится на основе структуры «node», которая содержит данные, ссылки на левого/правого потомка и высоту. В классе есть методы добавления элемента «insert», удаления элемента «remove», поиска элемента «search», вывода дерева в консоль «display» и расчет максимальной глубины «max_deep». Также в классе содержатся вспомогательные методы, которые осуществляют малый левый/правый поворот и большой левый/правый поворот.

```
class AVL
{
    struct node
    {
        int data;
        node* left;
        node* right;
        int height;
    };

    node* root;

    void makeEmpty(node* t)
    {
        if (t == NULL)
            return;
        makeEmpty(t->left);
        makeEmpty(t->right);
        delete t;
    }

    node* insert(int x, node* t)
    {
        if (t == NULL)
        {
            t = new node;
            t->data = x;
            t->height = 0;
            t->left = t->right = NULL;
        }
        else if (x < t->data)
        {
            t->left = insert(x, t->left);
            if (height(t->left) - height(t->right) == 2)
```

```
        {
            if (x < t->left->data)
                t = singleRightRotate(t);
            else
                t = doubleRightRotate(t);
        }
    }
    else if (x > t->data)
    {
        t->right = insert(x, t->right);
        if (height(t->right) - height(t->left) == 2)
        {
            if (x > t->right->data)
                t = singleLeftRotate(t);
            else
                t = doubleLeftRotate(t);
        }
    }

    t->height = max(height(t->left), height(t->right)) + 1;
    return t;
}

node* singleRightRotate(node*& t)
{
    node* u = t->left;
    t->left = u->right;
    u->right = t;
    t->height = max(height(t->left), height(t->right)) + 1;
    u->height = max(height(u->left), t->height) + 1;
    return u;
}

node* singleLeftRotate(node*& t)
{
    node* u = t->right;
    t->right = u->left;
    u->left = t;
    t->height = max(height(t->left), height(t->right)) + 1;
    u->height = max(height(t->right), t->height) + 1;
    return u;
}

node* doubleLeftRotate(node*& t)
{
    t->right = singleRightRotate(t->right);
    return singleLeftRotate(t);
}

node* doubleRightRotate(node*& t)
{
    t->left = singleLeftRotate(t->left);
    return singleRightRotate(t);
}

node* findMin(node* t)
{
    if (t == NULL)
        return NULL;
    else if (t->left == NULL)
        return t;
    else
        return findMin(t->left);
}
```

```
node* findMax(node* t)
{
    if (t == NULL)
        return NULL;
    else if (t->right == NULL)
        return t;
    else
        return findMax(t->right);
}

node* remove(int x, node* t)
{
    node* temp;

    // Element not found
    if (t == NULL)
        return NULL;

    // Searching for element
    else if (x < t->data)
        t->left = remove(x, t->left);
    else if (x > t->data)
        t->right = remove(x, t->right);

    // Element found
    // With 2 children
    else if (t->left && t->right)
    {
        temp = findMin(t->right);
        t->data = temp->data;
        t->right = remove(t->data, t->right);
    }
    // With one or zero child
    else
    {
        temp = t;
        if (t->left == NULL)
            t = t->right;
        else if (t->right == NULL)
            t = t->left;
        delete temp;
    }
    if (t == NULL)
        return t;

    t->height = max(height(t->left), height(t->right)) + 1;

    // If node is unbalanced
    // If left node is deleted, right case
    if (height(t->left) - height(t->right) == 2)
    {
        // right right case
        if (height(t->left->left) - height(t->left->right) == 1)
            return singleLeftRotate(t);
        // right left case
        else
            return doubleLeftRotate(t);
    }
    // If right node is deleted, left case
    else if (height(t->right) - height(t->left) == 2)
    {
        // left left case
        if (height(t->right->right) - height(t->right->left) == 1)
            return singleRightRotate(t);
        // left right case
```

```
            else
                return doubleRightRotate(t);
        }
        return t;
    }

    int height(node* t)
    {
        return (t == NULL ? -1 : t->height);
    }

    int getBalance(node* t)
    {
        if (t == NULL)
            return 0;
        else
            return height(t->left) - height(t->right);
    }

    void inorder(node* t)
    {
        if (t == NULL)
            return;
        inorder(t->left);
        cout << t->data << " ";
        inorder(t->right);
    }

    node* find(node* t, int x) {
        if (t == NULL)
            return NULL;
        else if (x < t->data)
            return find(t->left, x);
        else if (x > t->data)
            return find(t->right, x);
        else
            return t;
    }

public:
    AVL()
    {
        root = NULL;
    }

    ~AVL() {
        makeEmpty(root);
    }

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

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

    void display()
    {
        inorder(root);
        cout << endl;
    }
```

```
    void search(int x) {
        root = find(root, x);
    }
};
```

Функция «main» ~ состоит из цикла, в котором происходит генерация декартова дерева поиска и avl дерева на основе массива, заполненного случайными элементами, состоящего из 2^(10 + i) элементов, где i это номер серии теста. В каждой серии проводится 1000 операций поиска и удаления элемента изо всех деревьев, вычисляется максимальная глубина деревьев и замеряется суммарное время каждой операции. Полученные значения выводятся в файл «Time.txt»

Один из результатов работы программы:

```
Тест 2
Число элементов: 2048

<Массив случайных чисел>
Время операций (мс):

Вставка:
Treap: 9.39118
AVL: 6.04944

Максимальная глубина:
Treap: 31
AVL: 13

Среднее значение Максимальной глубины:
Treap: 24.96
AVL: 13

Поиск:
Treap: 0.925762
AVL: 0.25865

Удаление:
Treap: 5.32815
AVL: 0.277166
```

Рисунок 1 - Результат из файла "Time.txt"

	Время поиска элементов (мс) (общее)	
Элементов	Treap	AVL
1024	0,690	0,231
2048	0,926	0,259
4096	0,854	0,234
8192	0,989	0,241
16384	1,021	0,232
32768	1,324	0,239
65536	1,489	0,234
131072	1,672	0,240
262144	1,661	0,230

Таблица 1 – Время(мс) поиска 1000 случайных элементов в декартовом и AVL дереве.

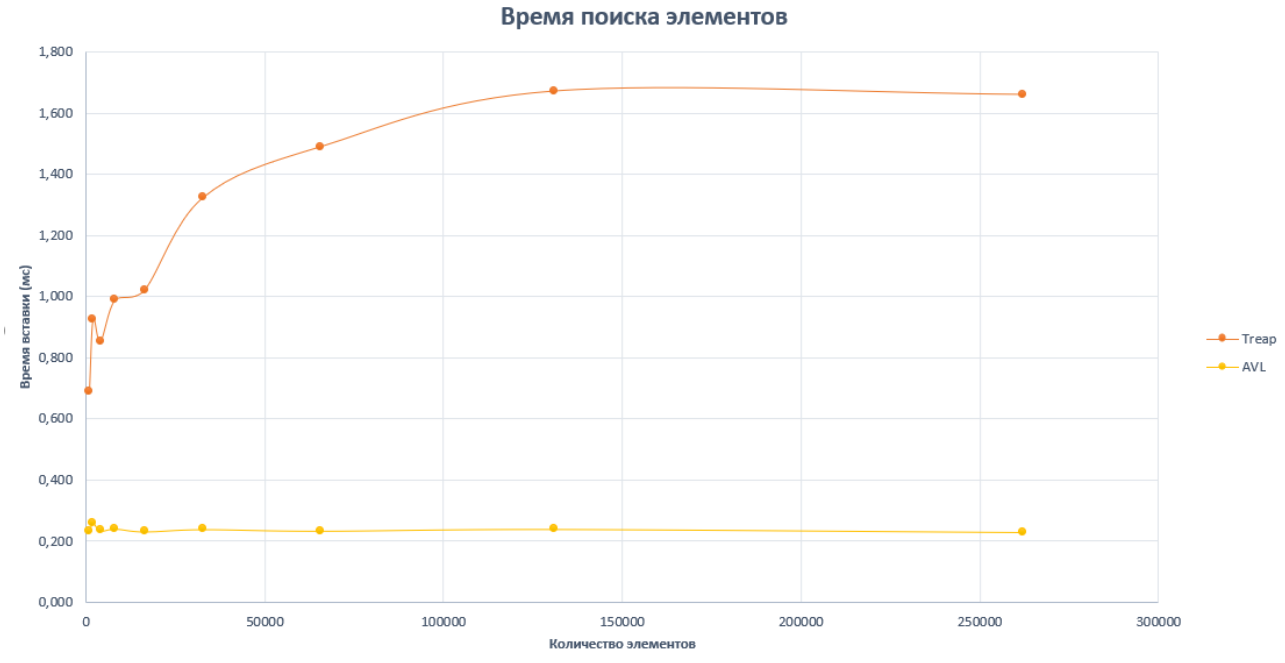

Рисунок 2 - График времени(мс) поиска 1000 случайных элементов в декартовом и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов.

	Время вставки элементов (мс) (общее)	
Элементов	Treap	AVL
1024	3,384	2,083
2048	9,391	6,049
4096	16,765	9,526
8192	37,522	20,917
16384	81,681	42,998
32768	194,224	96,155
65536	438,134	200,311
131072	957,016	393,618
262144	2081,250	768,515

Таблица 2 - Время(мс) вставки элементов в декартовом и AVL дереве.

Рисунок 2 - График времени(мс) вставки в декартовом и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов.

	Время удаления элементов (мс) (общее)	
Элементов	Treap	AVL
1024	4,361	0,257
2048	5,328	0,277
4096	4,911	0,257
8192	5,357	0,242
16384	5,969	0,245
32768	7,398	0,248
65536	8,661	0,251
131072	9,956	0,248
262144	11,195	0,244

Таблица 3 - Время(мс) удаления элементов в декартовом и AVL дереве.

Рисунок 3 - График времени(мс) удаления 1000 случайных элементов в декартовом и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов.

	Максимальная высота	
Элементов	Treap	AVL
1024	28,000	12,000
2048	31,000	13,000
4096	36,000	15,000
8192	36,000	16,000
16384	41,000	17,000
32768	42,000	18,000
65536	44,000	18,000
131072	49,000	18,000
262144	54,000	18,000

Таблица 4 - Максимальные высоты деревьев относительно количества элементов из 50 попыток.

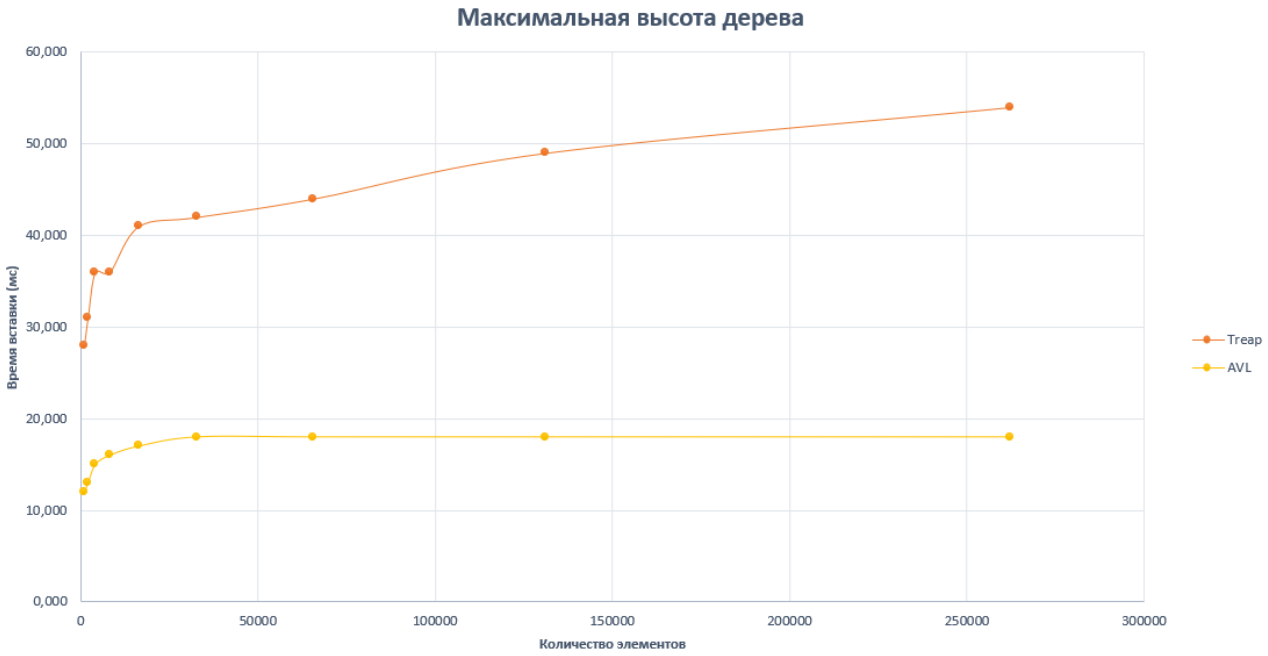

Рисунок 4 - График максимальной высоты в декартовом и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов за 50 попыток.

	Среднее распределение Максимальной высоты	
Элементов	Treap	AVL
1024	21,920	12,000
2048	24,960	13,000
4096	27,940	14,080
8192	30,280	15,200
16384	33,360	16,340
32768	36,360	17,040
65536	38,800	17,920
131072	42,400	18,000
262144	46,960	18,000

Таблица 5 – Среднее распределение максимальной высоты деревьев относительно количества элементов за 50 попыток

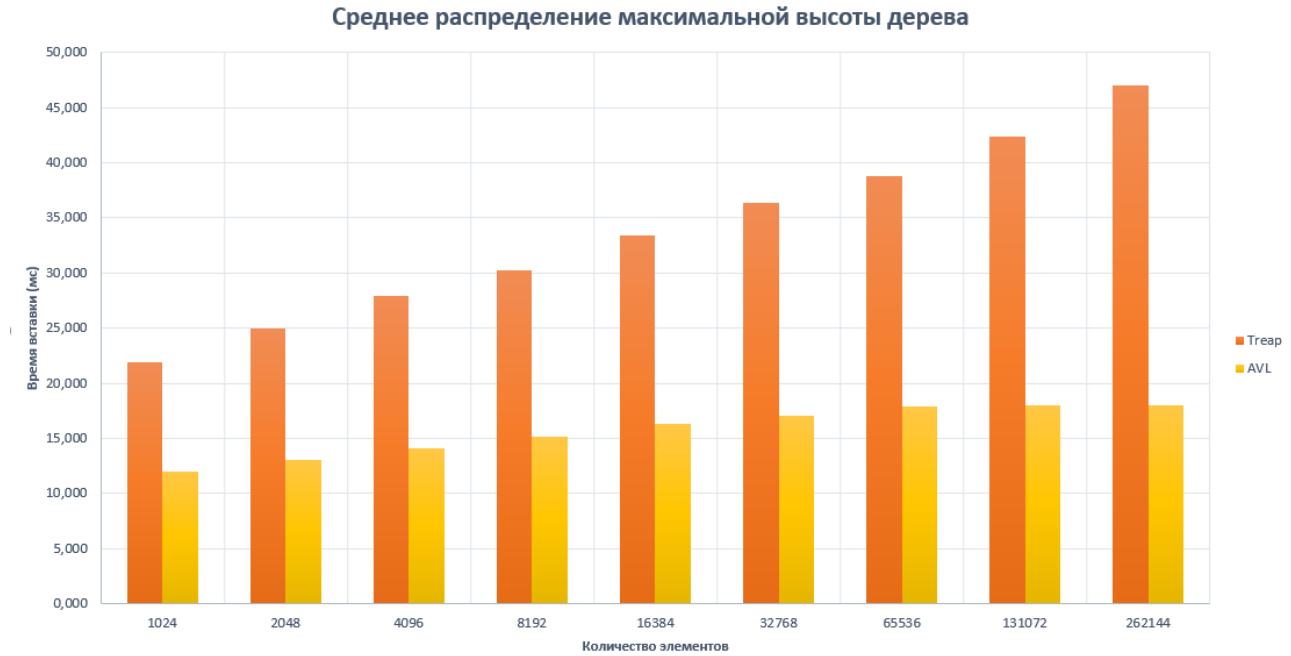

Рисунок 5 – Гистограмма среднего распределения максимальной высоты деревьев относительно количества элементов за 50 попыток

Заключение

В этой лабораторной работе я познакомился с такой структурой, как декартово дерево поиска и сравнил его с AVL деревом. Анализируя графики вставки элементов в деревья, заполненные из массива со случайным распределением, не трудно заметить, что вставка в декартово дерево занимает значительно больше времени (в 1.5 - 3 раза), чем вставка в AVL дерево, эта разница растет с увеличением количества элементов. Несмотря на то, что после вставки в AVL дерево выполняется его балансировка, как мы можем заметить, на выполнение таких операций в декартовом дереве, как соединение и разделение деревьев, требуется больше времени. Удаление элемента из декартова дерева занимает гораздо больше времени, чем удаление из AVL дерева. Это происходит по той же причине, что и при вставке элементов. Поиск в декартовом дереве так же занимает больше времени, чем поиск в AVL дереве. Высота AVL дерева практически всегда меньше высоты декартового дерева, поскольку суть AVL дерева прямо завязана на этом параметре и его построение идет таким образом, чтобы высота дерева была сбалансирована.

Таким образом, AVL дерево во всех операциях выигрывает у декартового дерева. Думаю, результаты тестов получились неточными, так как декартового дерево позволяет добавлять элементы с одинаковыми ключами, а в AVL дереве нет такой возможности.