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 №6

«Бинарное дерево поиска»

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: 18.04.2022

Москва
2022

Содержание

Описание задачи

В рамках лабораторной работы необходимо изучить и реализовать бинарное дерево поиска и его самобалансирующийся вариант в лице AVL дерева.

- Для проверки анализа работы структуры данных требуется провести 10 серий тестов.
- В каждой серии тестов требуется выполнять 20 циклов генерации и операций. При этом первые 10 работают с массивом заполненным случайным образом, во второй половине случаев, массив заполняется в порядке возрастания значений индекса, т.е. является отсортированным по умолчанию.
- Требуется создать массив состоящий из 2^(10 + i) элементов, где i это номер серии.
- Массив должен быть помещен в оба вариант двоичных деревьев. При этому замеряется время, затраченное на всю операцию вставки всего массива.
- После заполнения массива, требуется выполнить 1000 операций поиска по обоим вариантам дерева, случайного числа в диапазоне генерируемых значений, замерев время на все 1000 попыток и вычислив время 1 операции поиска.
- Провести 1000 операций поиска по массиву, замерить требуемое время на все 1000 операций и найти время на 1 операцию.
- После, требуется выполнить 1000 операций удаления значений из двоичных деревьев, и замерить время, затраченное на все операции, после чего вычислить время на 1 операцию.
- После выполнения всех серий тестов, требуется построить графики зависимости времени, затрачиваемого на операции вставки, поиска, удаления от количества элементов. При этом требуется разделить графики для отсортированного набора данных и заполненных со случайным распределением. Так же, для операции поиска, требуется так же нанести для сравнения график времени поиска для обычного массива.

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.

Выполнение задачи

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

Класс «BST» ~ занимается созданием бинарного дерева поиска. Дерево строится на основе структуры «node», которая содержит данные и ссылки на левого/правого потомка. В классе есть методы добавления элемента «insert», удаления элемента «remove», поиска элемента «search» и вывода дерева в консоль «display».

```
class BST {

    struct node {
        int data;
        node* left;
        node* right;
    };

    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->left = t->right = NULL;
        }
        else if (x < t->data)
            t->left = insert(x, t->left);
        else if (x > t->data)
            t->right = insert(x, t->right);
        return 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;
        if (t == NULL)
            return NULL;
        else if (x < t->data)
            t->left = remove(x, t->left);
        else if (x > t->data)
            t->right = remove(x, t->right);
        else if (t->left && t->right)
        {
            temp = findMin(t->right);
            t->data = temp->data;
            t->right = remove(t->data, t->right);
        }
        else
        {
            temp = t;
            if (t->left == NULL)
                t = t->right;
            else if (t->right == NULL)
                t = t->left;
            delete temp;
        }

        return t;
    }

    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:
    BST() {
        root = NULL;
    }

    ~BST() {
        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);
    }
        };
```

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» и вывода дерева в консоль «display». 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 бинарного дерева поиска и avl дерева на основе различных массивов (заполненного случайными элементами; отсортированного) состоящих из 2^(10 + i) элементов, где i это номер серии теста. В каждой серии проводится 1000 операций поиска и удаления элемента изо всех

деревьев и замеряется время каждой операции, как суммарное, так и на 1 операцию. Полученные значения выводятся в файл «Time.txt»

Один из результатов работы программы:

```
Тест 3
Число элементов: 4096

<Массив случайных чисел>
Время операций (мс):

Вставка:
BST: 3.12242
AVL: 9.87964

Поиск:
BST: 0.24945
AVL: 0.24777
Массив: 93.7362

Поиск (1 операция):
BST: 0.00024945
AVL: 0.00024777
Массив: 0.0937362

Удаление:
BST: 0.27582
AVL: 0.33004

Удаление (1 операция):
BST: 0.00027582
AVL: 0.00033004
```

Рисунок 1 - Результат из файла "Time.txt"

```
<Массив отсортированных чисел>
Время операций (мс):

Вставка:
BST: 249.588
AVL: 9.64772

Поиск:
BST: 0.24945
AVL: 0.24777
Массив: 53.2782

Поиск (1 операция):
BST: 0.00024945
AVL: 0.00024777
Массив: 0.0532782

Удаление:
BST: 0.28087
AVL: 0.40881

Удаление (1 операция):
BST: 0.00028087
AVL: 0.00040881
```

Рисунок 2 - Результат из файла "Time.txt" (продолжение)

	Время поиска элементов (мс) (общее)					
	Случайные элементы			Отсортированные элементы		
Элементов	BST	AVL	Массив	BST	AVL	Массив
1024	0,236	0,235	24,501	0,338	0,232	7,446
2048	0,241	0,240	44,087	0,387	0,239	13,073
4096	0,261	0,258	79,792	0,436	0,256	23,121
8192	0,286	0,277	140,733	0,786	0,283	41,319
16384	0,300	0,300	285,847	1,439	0,306	85,554

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

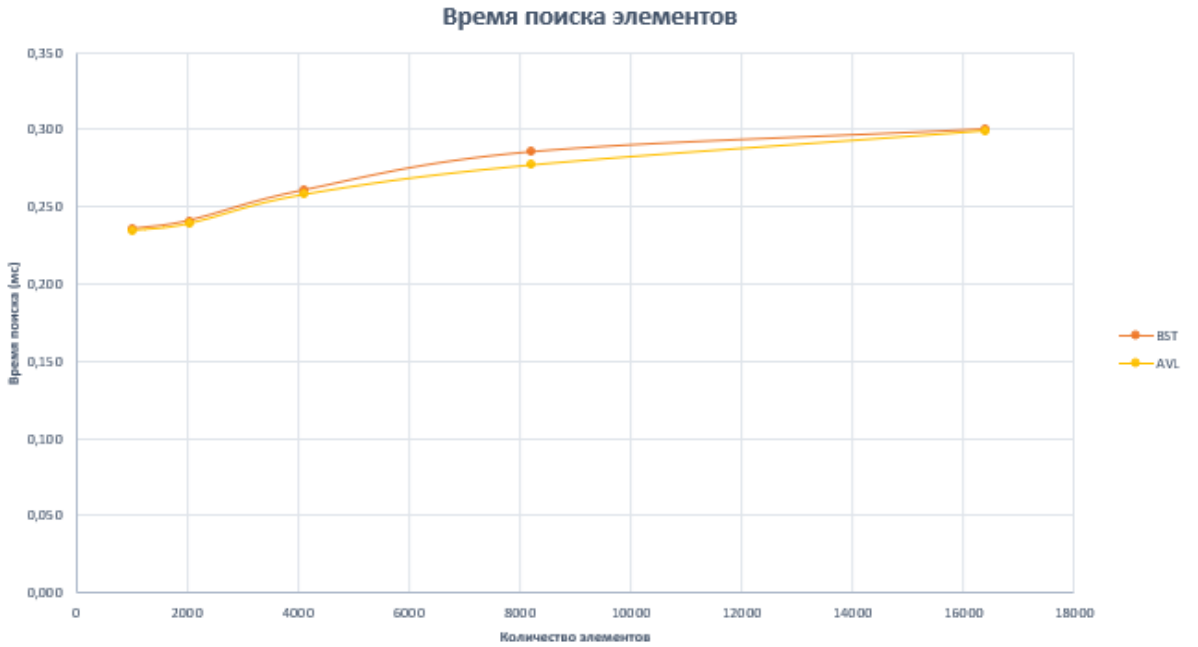

Рисунок 3 - График времени(мс) поиска 1000 случайных элементов в бинарном и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов.

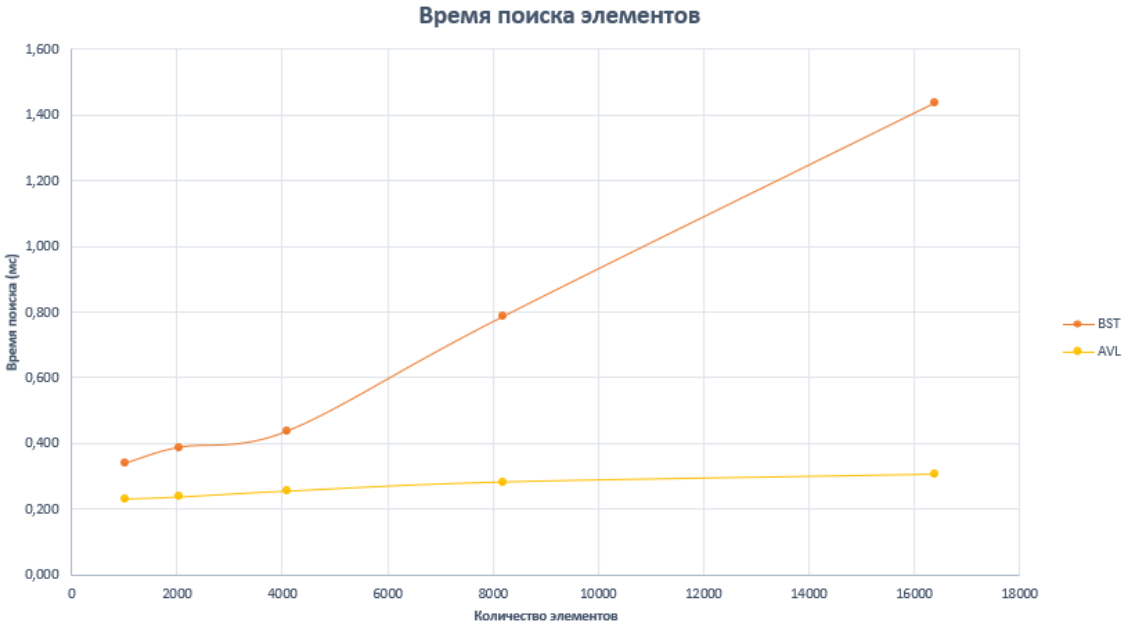

Рисунок 4 - График времени(мс) поиска 1000 случайных элементов в бинарном и AVL дереве (заполненных из отсортированного массива) относительно количества элементов.

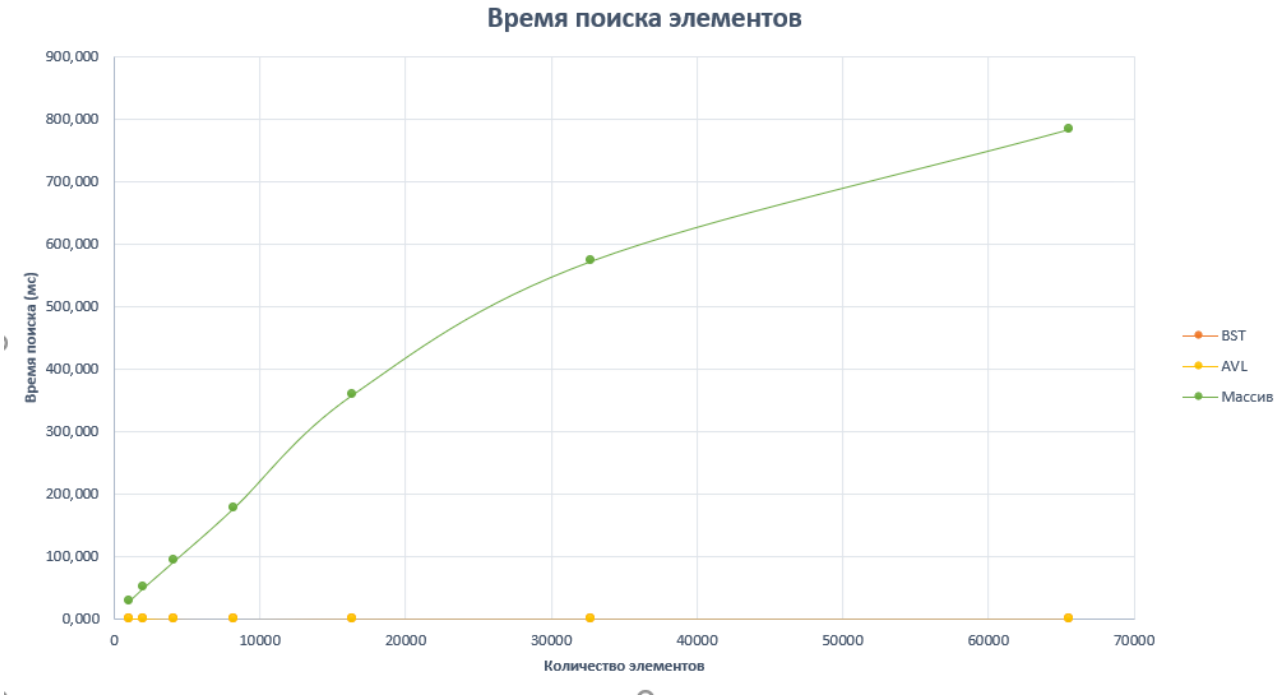

Рисунок 5 - График времени(мс) поиска 1000 случайных элементов в бинарном и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов в сравнение с поиском в обычном массиве.

	Время вставки элементов (мс) (общее)			
	Случайные элементы		Отсортированные элементы	
Элементов	BST	AVL	BST	AVL
1024	0,854	2,387	23,867	2,750
2048	1,767	5,518	75,629	5,459
4096	3,122	9,880	249,588	9,648
8192	5,507	18,475	1036,680	19,924
16384	12,629	41,104	5539,360	43,033
32768	28,052	88,422	44911,800	91,311
65536	51,455	179,549	216099,000	190,902

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

Рисунок 6 - График времени(мс) вставки в бинарном и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов.

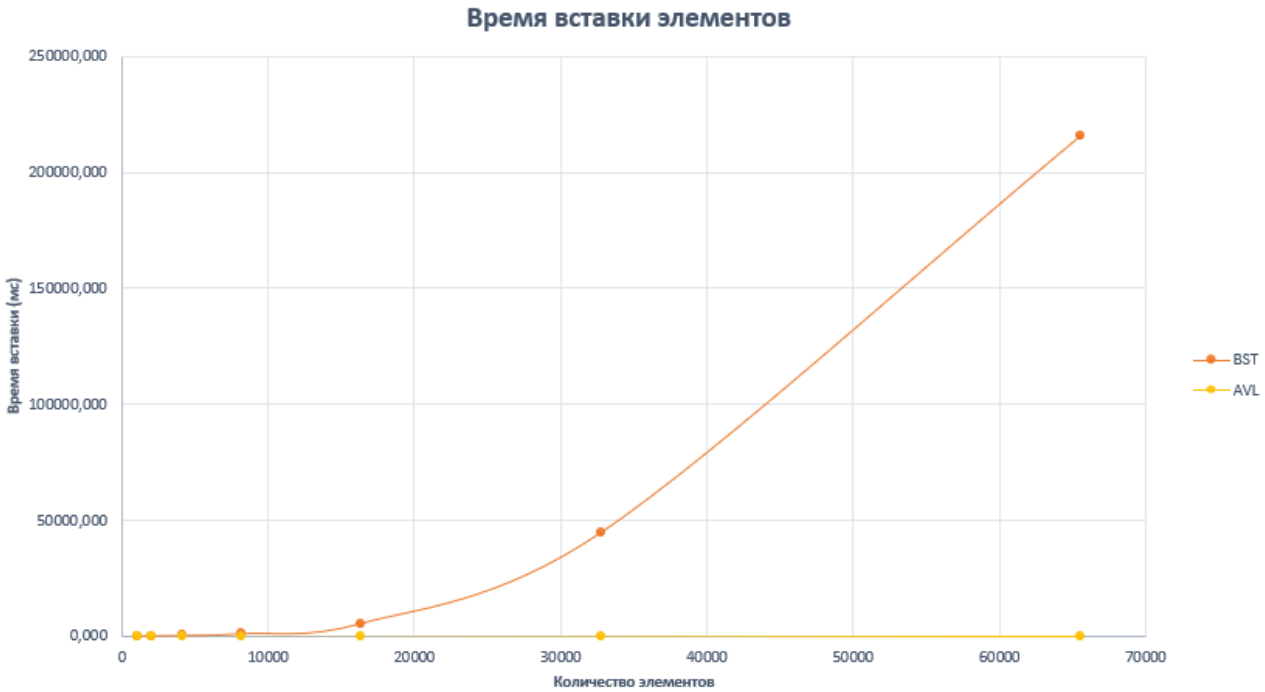

Рисунок 7 - График времени(мс) вставки в бинарном и AVL дереве (заполненных из отсортированного массива) относительно количества элементов.

	Время удаления элементов (мс) (общее)			
	Случайные элементы		Отсортированные элементы	
Элементов	BST	AVL	BST	AVL
1024	0,241	0,256	0,243	0,258
2048	0,226	0,242	0,227	0,243
4096	0,240	0,256	0,240	0,257
8192	0,252	0,271	0,251	0,272
16384	0,297	0,283	0,266	0,282
32768	0,343	0,374	0,342	0,353
65536	0,276	0,330	0,281	0,409

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

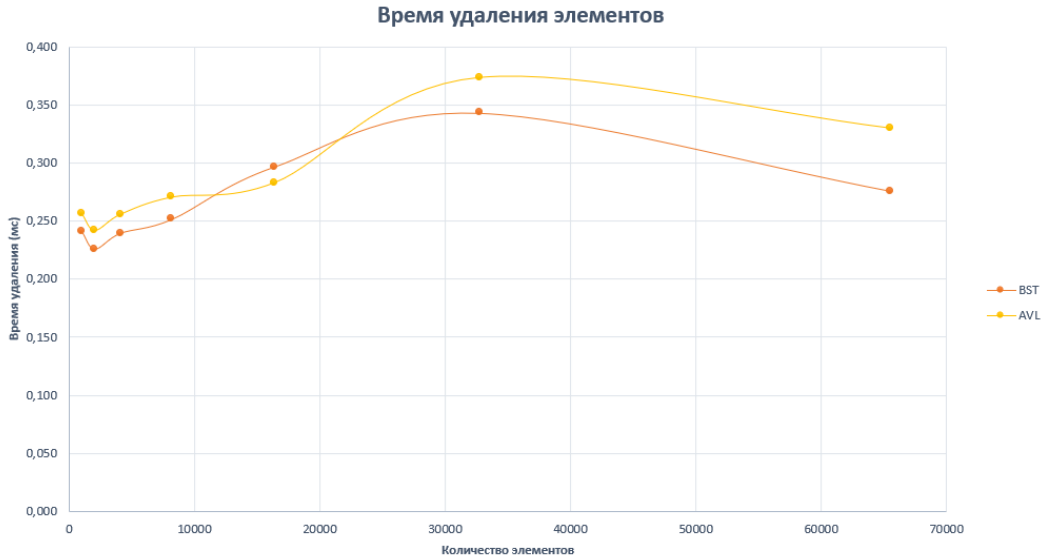

Рисунок 8 - График времени(мс) удаления 1000 случайных элементов в бинарном и AVL дереве (заполненных из массива со случайным распределением) относительно количества элементов.

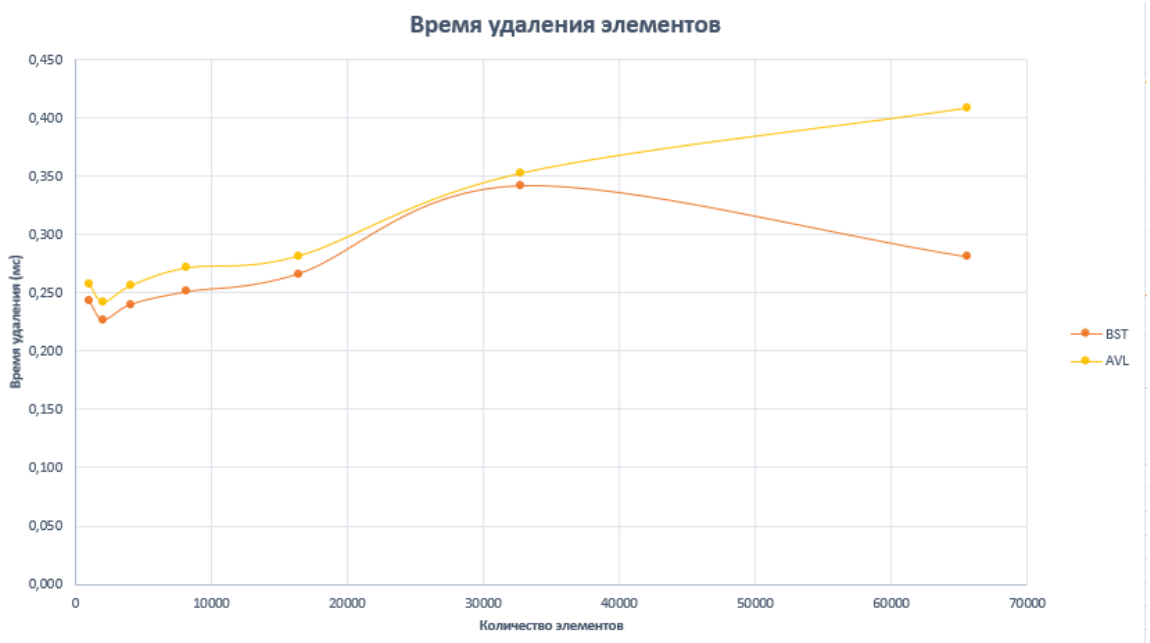

Рисунок 9- График времени(мс) удаления 1000 случайных элементов в бинарном и AVL дереве (заполненных из отсортированного массива) относительно количества элементов.

Заключение

В этой лабораторной работе я познакомился с такими структурами, как бинарное дерево поиска и его самобалансирующаяся версия в виде AVL дерева. Анализируя графики вставки элементов в бинарные деревья, заполненные из массива со случайным распределением, не трудно заметить, что вставка в AVL дерево занимает значительно больше времени (в 4 раза), чем вставка в обычное бинарное дерево поиска. Это легко объясняется, поскольку, после вставки в AVL дерево выполняется его балансировка. Однако, когда эти деревья заполняются из отсортированного массива, вставка в бинарное дерево поиска занимает колоссальное количество времени, чем вставка в AVL дерево. Это является худшим случаем для бинарного дерева поиска, когда высота дерева равна количеству вставляемых элементов. Время удаления из AVL дерева занимает немного больше времени, удаление из бинарного дерева поиска. Это происходит по той же причине, что и при вставке элементов, поскольку AVL дерево балансируется после каждого удаления элемента. Поиск в бинарном дереве поиска и AVL дереве проходит одинаково быстро, если деревья были заполнены из массива со случайным распределением элементов. Однако, когда эти деревья заполняются из отсортированного массива, поиск в бинарном дереве занимает гораздо большее количество времени (и приближен к линейному), чем поиск в AVL дереве. Это происходит по той же причине той же самой причине, как и при выполнении вставки. В целом, бинарные деревья отлично справляются со своей задачей, по сравнению с поиском в обычном массиве.