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 № 6

Выполнил студент группы .....КС-30..... Лихолат Полина Николаевна
Ссылка на репозиторий: .....https://github.com/MUCTR-IKT-CPP/Likholat_algorithms.git

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

Дата сдачи: 03.04.2023

Оглавление

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 метода/модели.

Бинарное дерево поиска (Binary Search Tree, BST) - это структура данных, которая представляет собой дерево, в котором каждая вершина содержит ключ (значение), и все ключи в левом поддереве меньше ключа в текущей вершине, а все ключи в правом поддереве больше ключа в текущей вершине. Ключи в BST должны быть уникальными.

BST поддерживает следующие операции:

- Вставка ключа в дерево
- Удаление ключа из дерева
- Поиск ключа в дереве

Операции вставки и удаления в BST выполняются за время O(log n) в среднем случае, где n - количество элементов в дереве. Операция поиска выполняется за время O(log n) в среднем случае и за время O(n) в худшем случае.

BST может быть реализован как рекурсивная структура данных, в которой каждая вершина содержит ссылки на ее левое и правое поддерево.

Вставка элемента в BST:

1. Если дерево пустое, создать новую вершину с заданным ключом и вернуть ее.
2. Если ключ меньше текущего ключа в вершине, рекурсивно вызвать функцию вставки для левого поддерева.
3. Если ключ больше текущего ключа в вершине, рекурсивно вызвать функцию вставки для правого поддерева.
4. Вернуть указатель на текущую вершину.

Удаление элемента из BST:

1. Если дерево пустое, вернуть NULL.
2. Если ключ меньше текущего ключа в вершине, рекурсивно вызвать функцию удаления для левого поддерева.
3. Если ключ больше текущего ключа в вершине, рекурсивно вызвать функцию удаления для правого поддерева.
4. Если ключ равен текущему ключу в вершине:
   - Если вершина не имеет потомков, удалить вершину и вернуть NULL.
   - Если вершина имеет только одного потомка, заменить вершину на ее потомка и вернуть указатель на потомка.
   - Если вершина имеет двух потомков, найти наименьший ключ в правом поддереве (или наибольший ключ в левом поддереве), заменить ключ текущей вершины на найденный ключ, и рекурсивно удалить вершину с найденным ключом из соответствующего поддерева.
5. Вернуть указатель на текущую вершину.

Поиск элемента в BST:

1. Если дерево пустое, вернуть NULL.
2. Если ключ меньше текущего ключа в вершине, рекурсивно вызвать функцию поиска для левого поддерева.
3. Если ключ больше текущего ключа в вершине, рекурсивно вызвать функцию поиска для правого поддерева.
4. Если ключ равен текущему ключу в вершине, вернуть указатель на текущую вершину.

BST может использоваться для решения широкого круга задач, таких как поиск наибольшего и наименьшего элемента, поиск наибольшего элемента, который меньше заданного, поиск наименьшего элемента, который больше заданного, и т.д.

Важным аспектом при работе с BST является балансировка дерева. Если дерево несбалансировано, операции вставки, удаления и поиска могут выполняться за время O(n), что делает BST неэффективным. Для балансировки BST используются различные алгоритмы, такие как AVL-деревья, красно-черные деревья, splay-деревья и др.

AVL-дерево является сбалансированным бинарным деревом поиска, в котором для каждой вершины высота ее двух поддеревьев различается не более чем на 1. Это свойство позволяет гарантировать, что операции вставки, удаления и поиска выполняются за время O(log n), где n - число вершин в дереве.

AVL-деревья являются одним из самых эффективных типов бинарных деревьев поиска, поскольку гарантируют логарифмическую сложность для основных операций, что делает их очень полезными для использования в различных приложениях и алгоритмах.

Выполнение задачи.

BST может быть реализован как рекурсивная структура данных, в которой каждая вершина содержит ссылки на ее левое и правое поддерево.

Вставка элемента в BST:

1. Если дерево пустое, создать новую вершину с заданным ключом и вернуть ее.
2. Если ключ меньше текущего ключа в вершине, рекурсивно вызвать функцию вставки для левого поддерева.
3. Если ключ больше текущего ключа в вершине, рекурсивно вызвать функцию вставки для правого поддерева.
4. Вернуть указатель на текущую вершину.

Удаление элемента из BST:

1. Если дерево пустое, вернуть NULL.
2. Если ключ меньше текущего ключа в вершине, рекурсивно вызвать функцию удаления для левого поддерева.
3. Если ключ больше текущего ключа в вершине, рекурсивно вызвать функцию удаления для правого поддерева.
4. Если ключ равен текущему ключу в вершине:
   - Если вершина не имеет потомков, удалить вершину и вернуть NULL.
   - Если вершина имеет только одного потомка, заменить вершину на ее потомка и вернуть указатель на потомка.

- Если вершина имеет двух потомков, найти наименьший ключ в правом поддереве (или наибольший ключ в левом поддереве), заменить ключ текущей вершины на найденный ключ, и рекурсивно удалить вершину с найденным ключом из соответствующего поддерева.

5. Вернуть указатель на текущую вершину.

Поиск элемента в BST:

1. Если дерево пустое, вернуть NULL.
2. Если ключ меньше текущего ключа в вершине, рекурсивно вызвать функцию поиска для левого поддерева.
3. Если ключ больше текущего ключа в вершине, рекурсивно вызвать функцию поиска для правого поддерева.
4. Если ключ равен текущему ключу в вершине, вернуть указатель на текущую вершину.

AVL-дерево является сбалансированным бинарным деревом поиска, в котором для каждой вершины высота ее двух поддеревьев различается не более чем на 1.

Будем представлять узлы АВЛ-дерева следующей структурой:

```
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; }
};
```

Поле key хранит ключ узла, поле height — высоту поддерева с корнем в данном узле, поля left и right — указатели на левое и правое поддеревья. Простой конструктор создает новый узел (высоты 1) с заданным ключом k.

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.

Основной механизм поддержания баланса в AVL-дереве - это операции поворота. Два основных типа поворотов - левый и правый - используются для увеличения или уменьшения глубины поддеревьев.

При вставке новой вершины в AVL-дерево сначала выполняется стандартная операция вставки, а затем проверяется баланс дерева. Если для какой-то вершины нарушается условие баланса, то выполняются соответствующие операции поворота, чтобы восстановить баланс.

```
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;
}
```

Левый поворот является симметричной копией правого:

```
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;
}
```

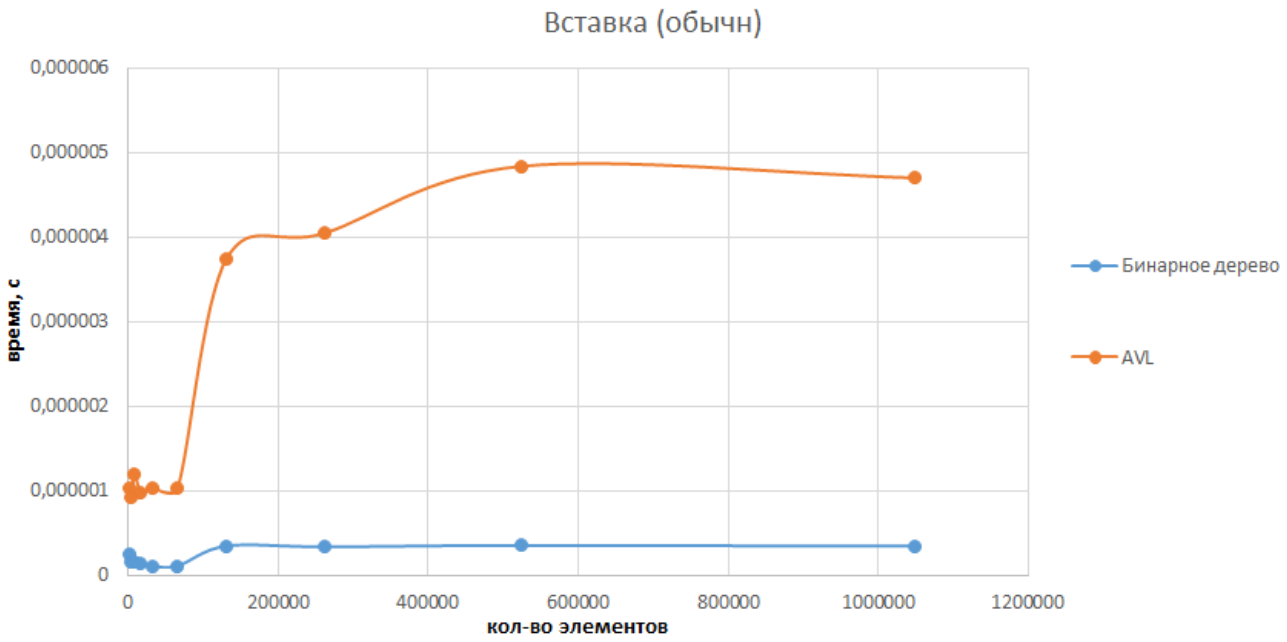

Рисунок 1.1 График зависимости времени вставки от кол-ва элементов

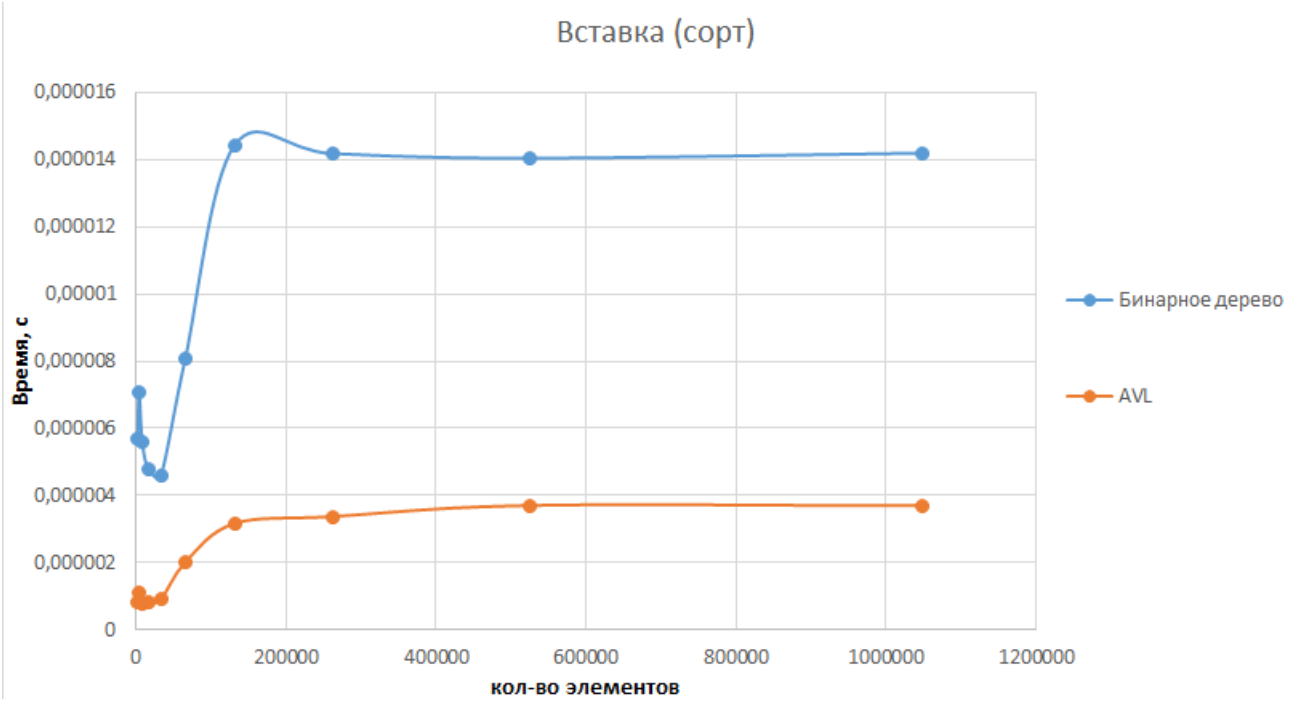

Рисунок 1.2. График зависимости времени вставки от кол-ва элементов для сортированных данных

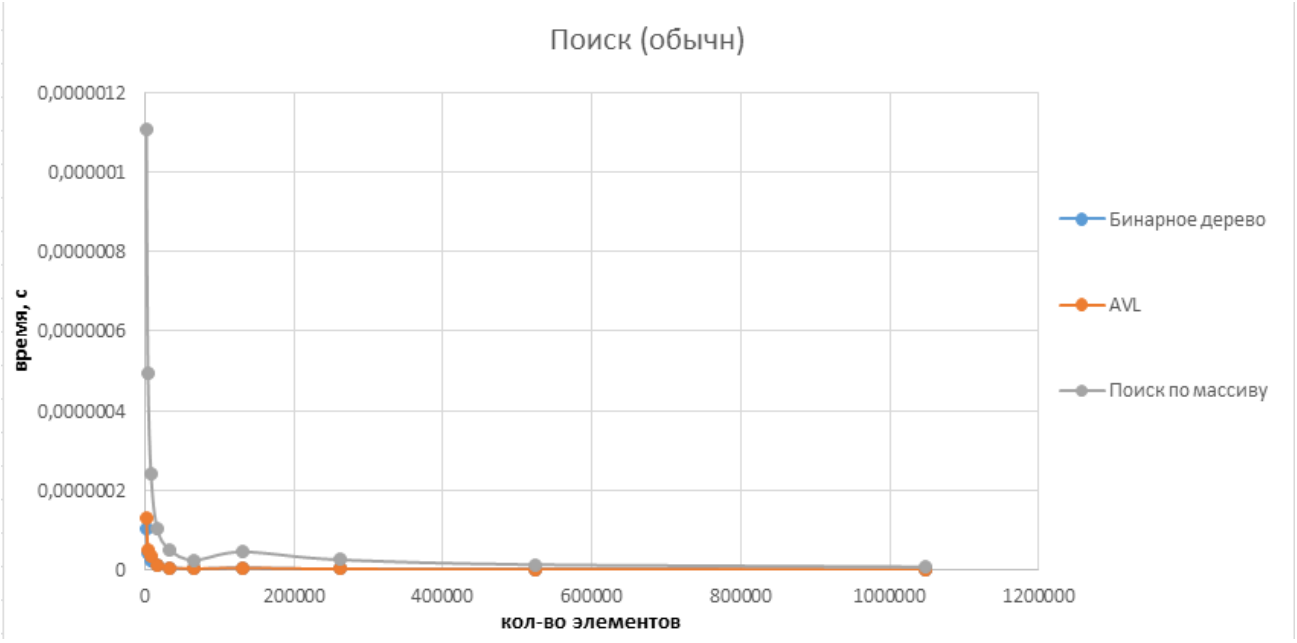

Рисунок 1.3. График зависимости времени поиска от кол-ва э

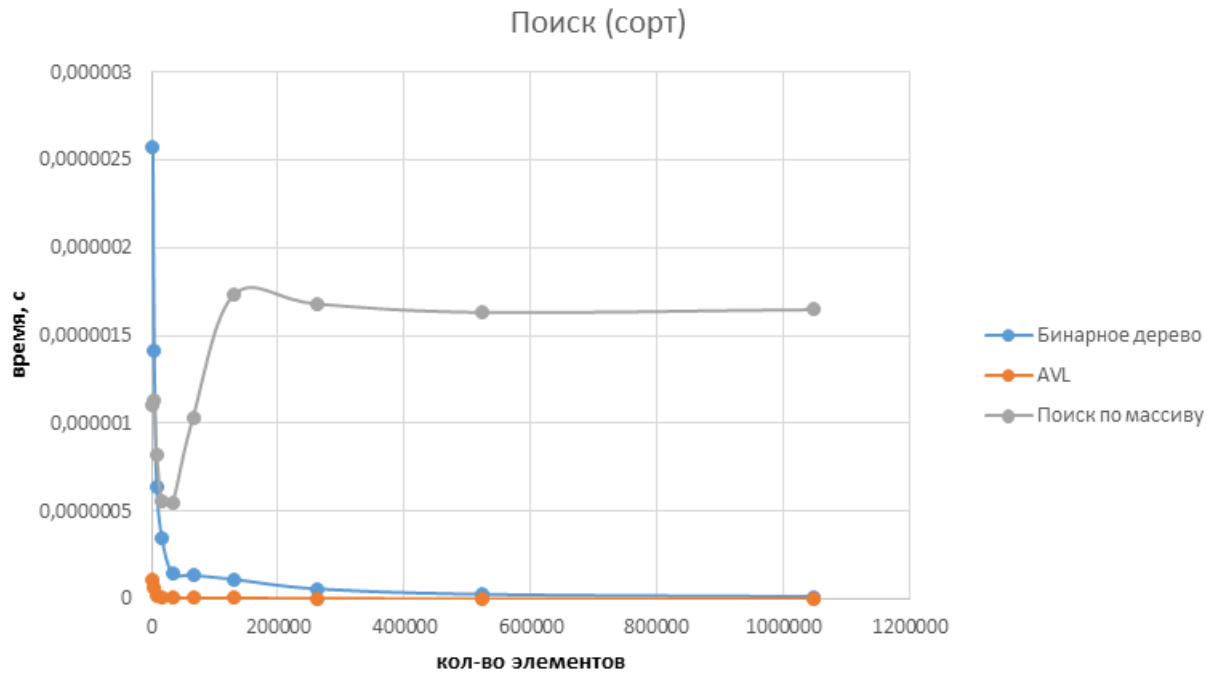

Рисунок 1.4. График зависимости времени поиска от кол-ва элементов для сортированных данных

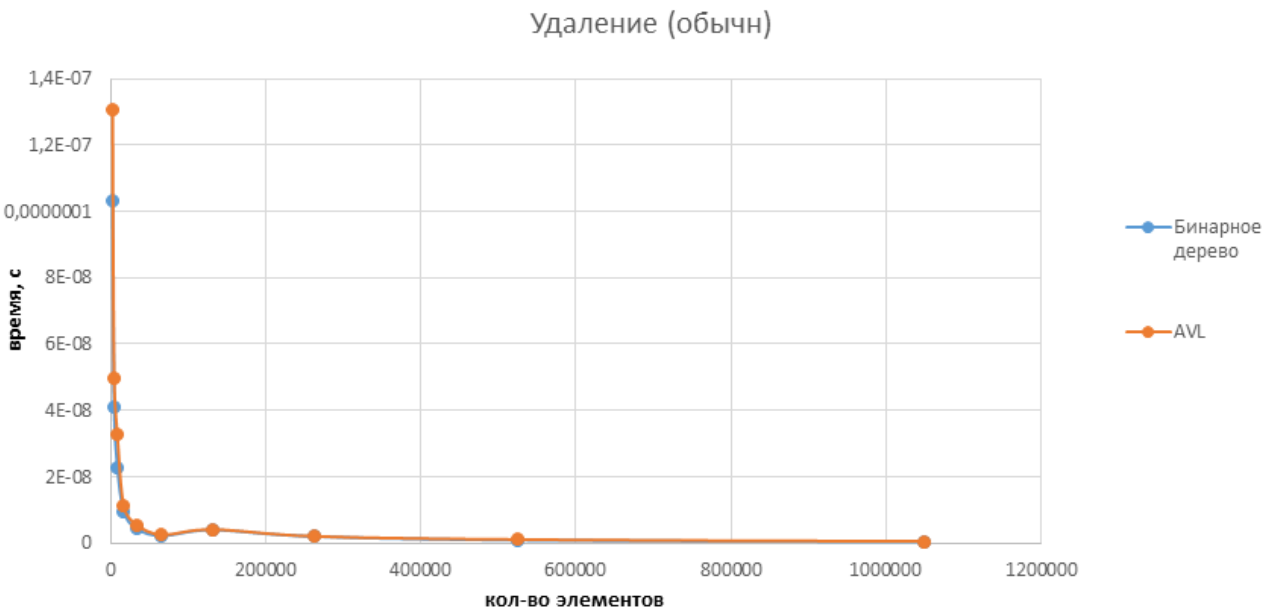

Рисунок 1.5. График зависимости времени удаления от кол-ва элементов

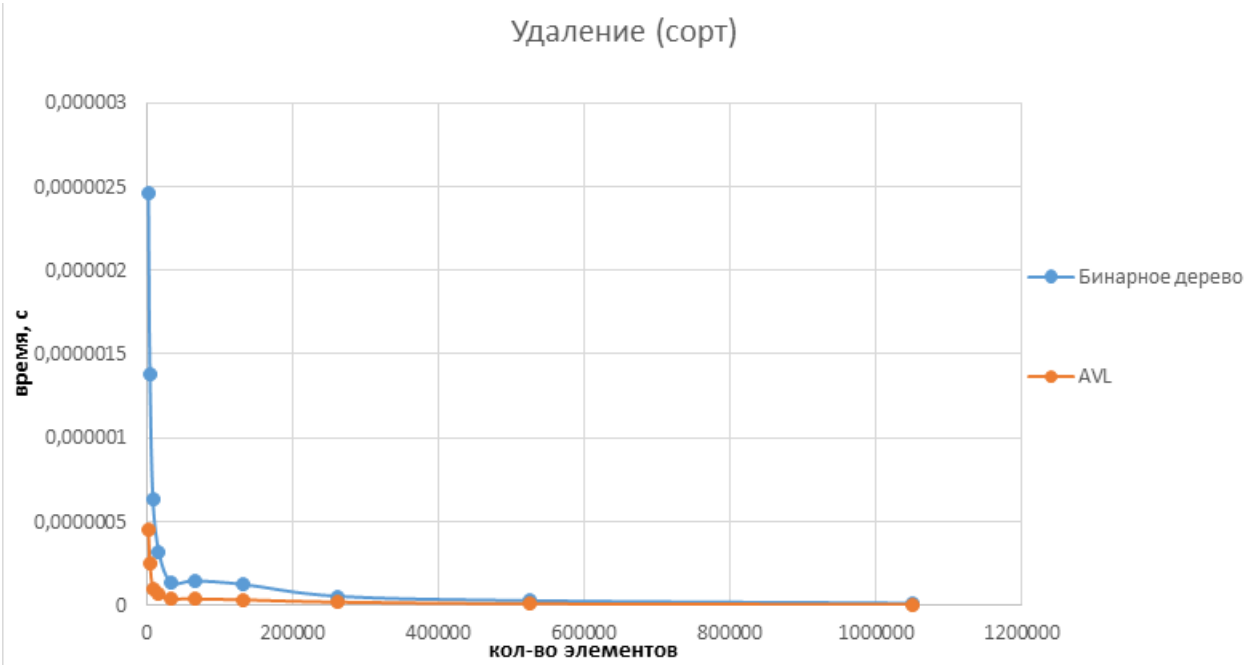

Рисунок 1.6. График зависимости времени удаления от кол-ва элементов для сортированных данных

Заключение.

Бинарное и AVL деревья являются важными структурами данных для хранения и быстрого поиска элементов в упорядоченных множествах. Оба дерева имеют свои преимущества и недостатки.

Бинарное дерево является простой и быстрой структурой данных, но может иметь неоптимальную высоту в случае несбалансированного расположения элементов. Это может привести к ухудшению времени выполнения операций поиска, вставки и удаления.

AVL дерево является самобалансирующейся структурой данных, которая поддерживает оптимальную высоту дерева. Это гарантирует лучшую производительность при выполнении операций поиска, вставки и удаления элементов.

Однако AVL дерево имеет некоторые недостатки, связанные с увеличенным объемом памяти для хранения дополнительных балансировочных информационных битов, а также с более сложной реализацией и обработкой операций вставки, удаления и перебалансировки.

В целом, выбор между бинарным и AVL деревом зависит от конкретной задачи и требований к производительности. Если данных мало и дерево не будет часто меняться, то бинарное дерево может быть лучшим выбором. Если же требуется быстрый поиск и обработка больших объемов данных, то следует рассмотреть использование AVL дерева.