ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 7
ДЕКАРТОВО ДЕРЕВО

Оглавление

Описание задачи.

Изучить и реализовать 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.

Задания со звездочкой = + 5 дополнительных первичных баллов:

- Аналогичная серия тестов и сравнение ее для отсортированного заранее набора данных

Описание метода/модели.

Декартово дерево

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.

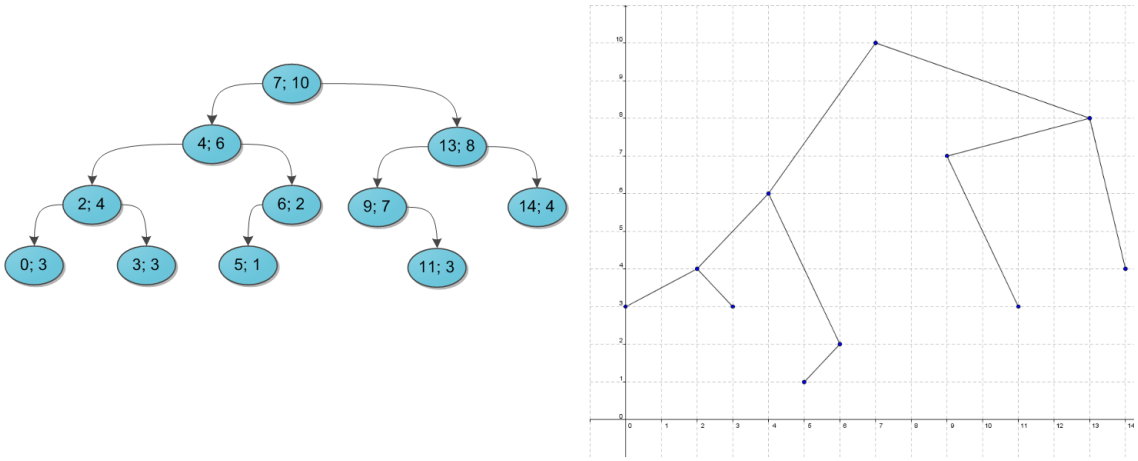

Рис.1. Декартово дерево.

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 логорефмической.

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.

Merge

Для работы этой операции у нас должно быть два поддерева L и R таких, что бы каждый ключ из L был строго меньше любого ключа из R и наоборот, любой ключ из R был строго больше любого ключа их L. Как объеденить два дерева? Тут на помощь приходит приоритет, и получается следующий набор действий:

1. Сравнить приоритеты корней L и R. Больший сделать корнем нового дерева.

    - Если корнем стал корень L, то все его левое поддерево остается на своем месте, так как оно строго меньше корня.
2. 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 отсеется на своем месте, так как оно строго больше корня.
    - 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.

Split

Для работы операции, ей на вход подается декартово дерево T и ключ x который 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?

- Сравниваем корень дерева со значением ключа разделения.
- Если корень T меньше ключа, по которому происходит деление, то именно этот узел и будет является корнем результата деления L, а все левое поддерево будет являться частью L, так как оно меньше T, которая меньше x.
    - 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.
4. 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.
    - 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.

Вставка

Для обеспечения операции вставки мы воспользуемся операциями Merge и Split.

1. Создадим отдельное дерево M содержащее только вставляемое значение назначив ему случайный приоритет.
2. Разделим дерево на L и R операцией Split по значению вставляемого ключа. Таким образом мы получим значения, которые строго больше и строго меньше чем наше вставляемое значение.
3. Объединим операцией Merge дерево вставляемого значения с M с L. Мы можем так сделать так как L строго меньше чем единственный узел M.
4. Объединим операцией Merge полученное на предыдущем шаге дерево M+L с деревом R. Мы можем так сделать, так как все что в R строго больше чем объединение M+L.

Данная операция имеет логарифмическую асимптотическую сложность, но большую константу, чем аналогичная вставка для AVL дерева. Что будет заметно только на больших значениях.

Удаление

Для обеспечения операции удаления мы воспользуемся операциями Merge и Split. При обработке удаления, важно учитывать в какой стороне находятся равные корню ключи. Алгоритм будет приведен для левого их расположения.

1. Разделим дерево по ключу x – 1. Таким образом все элементы, меньше либо равные x – 1 отправились в L, а удаляемый ключ в R.
2. Разделим R полученный на прошлом шаге по ключу X. Таким образом все элементы больше чем удаяемый элемент попадут в R, а то что требуется удалить попадет в L.
3. Объединим результаты L с операции 1 и R с операции 2. Таким образом мы получим новое дерево, которое не будет содержать значений равных удаляемому ключу.

В случае если бы равенство ключей решалось бы через правое расположение, то следовало бы проводить разбиение по-другому, сначала x + 1, потом x, и отбрасывать правую из 2 шага.

Выполнение задачи.

Программа для получения тестовых данных была написана на C#.

Код:

```
namespace Treap_Lib
{
    /// <summary>
    /// Класс Декартова дерева
    /// </summary>
    public class Treap
    {
        /// <summary>
        /// Узлы Декартова дерева
        /// </summary>
        public class Node
        {
            public int x;//ключ
            public int y;//приоритет

            public Node Left;//левый потомок
            public Node Right;//правый потомок


            public Node(int x, int y, Node left = null, Node right = null)
            {
                this.x = x;
                this.y = y;
                this.Left = left;
                this.Right = right;
            }
            /// <summary>
            /// Рекурсивное Соедининие двух Поддеревьем
            /// </summary>
            /// <param name="L">Левое</param>
            /// <param name="R">Правое</param>
            /// <returns>Соединенное дерево</returns>
            public static Node Merge(Node L, Node R)
            {
                //провекра на потомков
                if (L == null) return R;
                if (R == null) return L;
                //поиск
                if (L.y > R.y)//если левый приоритет больше
                {
                    var newR = Merge(L.Right, R);//ищем новое правое поддерево
                    return new Node(L.x, L.y, L.Left, newR);
                }
                else//если правый приоритет больше
                {
                    var newL = Merge(L, R.Left);//ищем новое левое поддерево
                    return new Node(R.x, R.y, newL, R.Right);
                }
            }
            /// <summary>
            /// Рекурсивное Разделение дерева на два поддерева, где все значения одно меньше
всех знач другого
            /// </summary>
            /// <param name="x">Ключ для сравнения</param>
            /// <param name="L">Возвращаемое Левое поддерево</param>
            /// <param name="R">Возвращаемое Правое поддерево</param>
            public void Split(int x, out Node L, out Node R)
            {
                Node newTree = null;//создание нового дерева
                if (this.x <= x)//если у дерева T ключ меньше или равен x
                {
                    if (Right == null)
```

```
                    R = null;
                else
                    Right.Split(x, out newTree, out R);
                L = new Node(this.x, y, Left, newTree);//левое поддерево, где исключены R,
превыщающие x
            }
            else//если у дерева T ключ больше x
            {
                if (Left == null)
                    L = null;
                else
                    Left.Split(x, out L, out newTree);
                R = new Node(this.x, y, newTree, Right);//правое поддерево, где исключены
L, превыщающие x
            }
        }
        /// <summary>
        /// Рекурсивное Добавление элемента
        /// </summary>
        /// <param name="x">Добавляемый элемент</param>
        /// <returns>Дерево с добавленным элементом</returns>
        public Node Add(int x)
        {
            Random rand = new Random();
            Node l, r;
            Split(x, out l, out r);//разделение на две половины
            Node m = new Node(x, rand.Next());//дерево m из единственной вершины с новым
ключом
            return Merge(Merge(l, m), r);//поочередное объединение с l, а потом с r
        }
        /// <summary>
        /// Рекурсивное Удаление элемента
        /// </summary>
        /// <param name="x">Удаляемый элемент</param>
        /// <param name="root">Дерево</param>
        /// <returns>Дерево с удаленным элементом</returns>
        public Node Remove(int x, Node root)
        {
            Node l, m, r;
            Split(x - 1, out l, out r);//разделение дерева на две области так, чтобы
направо перенесся удаляемый элемент
            if (r != null)
            {
                r.Split(x, out m, out r);//Разделение правой части на две, где из-за x-1,
нам вернется r с удаленным ключом
                return Merge(l, r);//соединение l и r
            }
            else return root;
        }

    }
    public Node root;
    /// <summary>
    /// Конструктор для Декартова дерева
    /// </summary>
    /// <param name="x">ключ</param>
    /// <param name="y">приоритет</param>
    /// <param name="left">левый потомок</param>
    /// <param name="right">правый потомок</param>
    public Treap(int x, int y, Node left = null, Node right = null)
    {
        this.root = new Node(x, y, left, right);
    }
    /// <summary>
    /// Добавление нового ключа в дерево
    /// </summary>
    /// <param name="x">ключ</param>
    public void Add(int x)
    {
```

```
    this.root = this.root.Add(x);
}
/// <summary>
/// Удаление ключа из дерева
/// </summary>
/// <param name="x">ключ</param>
public void Remove(int x)
{
    this.root = this.root.Remove(x, this.root);
}
/// <summary>
/// Вывод дерева в консоль
/// </summary>
public void PrintTree()
{
    if (root == null)
    {
        Console.WriteLine("Дерево пусто");
        return;
    }
    int depth = 0;
    printTreeOrder(root, depth);
    Console.WriteLine();
}
/// <summary>
/// Вывод дерева слева-направо
/// </summary>
/// <param name="current"></param>
private void printTreeOrder(Node current, int depth)
{
    if (current != null)
    {
        //Console.Write("\n");
        depth++;
        printTreeOrder(current.Left, depth);
        Console.Write($"{depth}-({current.x}:{current.y}) ");
        printTreeOrder(current.Right, depth);

    }
}

/// <summary>
/// Метод нахождения узла
/// </summary>
/// <param name="el">узел</param>
public void Find(int el)
{
    Node find = Find(root, el);
    if (find != null && find.x == el)
    {
        //Console.WriteLine("{0} was found!", el);
    }
    else
    {
        //Console.WriteLine("Nothing found!");
    }
}
/// <summary>
/// Рекурсивный метод поиска узла в дереве
/// </summary>
/// <param name="current">Текущий узел</param>
/// <param name="target">Разыскиваемый узел</param>
/// <returns></returns>
private Node Find(Node current, int target)
{
    if (current == null) return null;
    if (target < current.x)
    {
        if (target == current.x)
```

```
            {
                return current;
            }
            else
                return Find(current.Left, target);
        }
        else
        {
            if (target == current.x)
            {
                return current;
            }
            else
                return Find(current.Right, target);
        }

    }
    /// <summary>
    /// Нахождение глубины Деррева
    /// </summary>
    /// <returns>Максимальная глубина</returns>
    public int findDepth()
    {
        if (root == null)
        {
            //Console.WriteLine("Дерево пусто");
            return -1;
        }
        int depth = 0;
        int depth_max = 0;
        depthTreeOrder(root, depth, ref depth_max);
        return depth_max;

    }
    /// <summary>
    /// Рекурсивное прохождение по массиву в поисках глубины
    /// </summary>
    /// <param name="current">текущий узел</param>
    /// <param name="depth">счётчик глубины</param>
    /// <param name="depth_max">максимальная глубина</param>
    private void depthTreeOrder(Node current, int depth, ref int depth_max )
    {
        if (current != null)
        {
            depth++;
            if (depth_max < depth) depth_max = depth;
            depthTreeOrder(current.Left, depth, ref depth_max);
            depthTreeOrder(current.Right, depth, ref depth_max);

        }
    }

    /// <summary>
    /// Нахождение глубины для каждой ветки дерева
    /// </summary>
    /// <returns>Список с найденными глубинами</returns>
    public List<int> findDepthEach()
    {
        if (root == null)
        {
            //Console.WriteLine("Дерево пусто");
            return null;
        }
        List<int> depth_arr = new List<int>() ;
        int depth = 0;
        depthEachTreeOrder(root, depth, ref depth_arr);
        return depth_arr;

    }
```

```
        /// <summary>
        /// Рекурсивное нахэождение глубины для каждой ветки дерева
        /// </summary>
        /// <param name="current">теущий узел</param>
        /// <param name="depth">счётчик глубины</param>
        /// <param name="depth_arr">список для сохранения значений</param>
         private void depthEachTreeOrder(Node current, int depth, ref List<int> depth_arr)
         {
             if (current != null)
             {
                 depth++;
                 depthEachTreeOrder(current.Left, depth, ref depth_arr);
                 if(!depth_arr.Contains(depth)) depth_arr.Add(depth);
                 depthEachTreeOrder(current.Right, depth, ref depth_arr);

             }
         }
    }
}
```

Реализация:

1)Декартово дерево представляет собой класс Treap с полями root(всё дерево) и подклассом Node с полями x(хранит значение узла), y(значение приоритета), left (хранит левого потомка), right(правый потомок).

2)Также в классе Node содержатся методы Merge, принимающий узлы L и R и по алгоритму:

- Сравнивает приоритеты корней L и R. Больший делает корнем нового дерева.
- Если корнем стал корень L, то все его левое поддерево остается на своем месте, так как оно строго меньше корня.
    - 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.
- Если корнем стал корень R, то все его правое поддерево отсеется на своем месте, так как оно строго больше корня.
    - 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.

3)Функция Split(), принимающая разыскиваемое значение, левый L, правый R узел и по алгоритму и возвращающая разделенные половины, где в одной все значения меньше, чем в другой:

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

4)Функция Add() вставляет принимаемо значение в дерево.

5)Функция Remove удаляет принимаемое значения из дерева и возвращает дерево без него.

6)PrintTree проходит рекурсирвно в функции printTreeOrder по всем ветвям слева-направо и выводит.

7)Функция Find ищет отправляемый ей в качестве параметра элемент в дереве, рекурентно перебирая узлы слева-направо.

8) Функция FindDepth считает высоту дерева, рекурсивно пробегаясь по нему в функции depthTreeOrder слева-направо.

9)И функция depthEachTree считает кол-во поддеревьев у всех узлов рекурсивно проходя слева-направо по дереву в функции depthEachTreeOrder

Результаты:

1)Демонстрация работы дерева.

Вывод Декартова дерева с добавленными элементами в порядке 5, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9:
Высота-(ключ: приоритет).

```
                                    9 :1670338996
                              6 : 1433183067--
                        0 : 1292438075 – 7 : 396846232
                            --3: 1241704732       --8 : 10469208
                        2: 304226619 –5: 709448114
                  1: 109261227 --       4: 299669463 –
                                              --5:17
```

Вывод того же дерева с удаленной 1,6:

```
                                    9 :1670338996
                              0 : 1292438075 --
                                        --3: 1241704732
                                  2: 304226619--5: 709448114
                                            4: 299669463 –7 : 396846232
                                                  --5:17            --8 : 10469208
```

2) Замерил время вставки несортированного Декартова Дерева в секундах

Таблица 1

Время вставки в несортированное декартово дерево в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000897	0.000004892	0.000001691
11	0.000000963	0.000019216	0.000002193
12	0.000001047	0.000006283	0.000002298
13	0.000001187	0.000013214	0.000003756
14	0.000001296	0.000011453	0.000003974
15	0.000001423	0.000010967	0.000004167
16	0.000001606	0.000046077	0.000009276
17	0.000001819	0.000093888	0.000011279
18	0.000001956	0.000057948	0.000010868

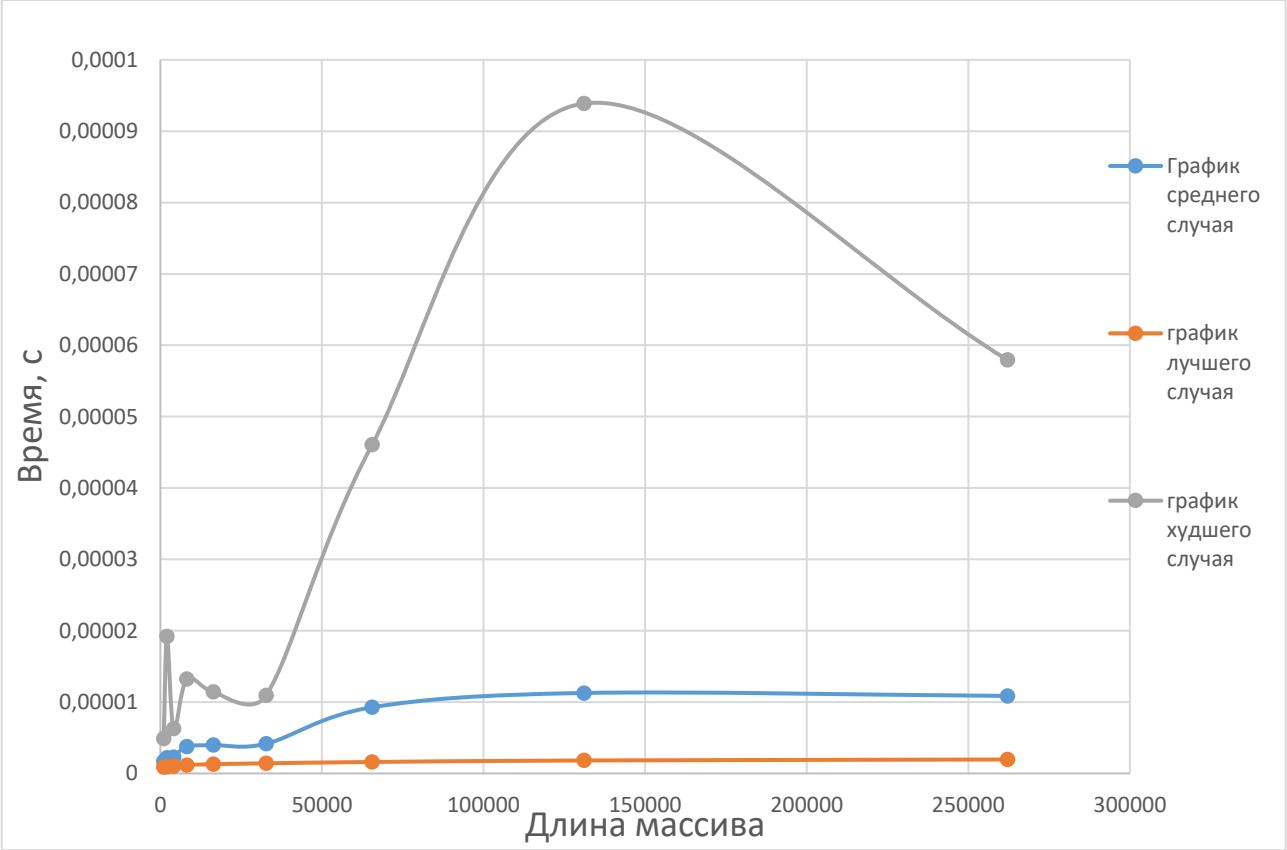

Рис.2. График времени, за которое происходит вставка в несортированное декартово дерево

3) Замерил время вставки сортированного Декартова Дерева в секундах

Таблица 2

Время вставки в отсортированное заранее декартовое дерево в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000422	0.000006195	0.000001123
11	0.000000853	0.000021533	0.000002264
12	0.000000894	0.000005696	0.000001200
13	0.000000911	0.000032817	0.000006615
14	0.000000864	0.000030444	0.000005136
15	0.000000928	0.000027588	0.000005844
16	0.000000928	0.000056421	0.000008196
17	0.000000911	0.000040713	0.000006823
18	0.000000987	0.000048866	0.000010009

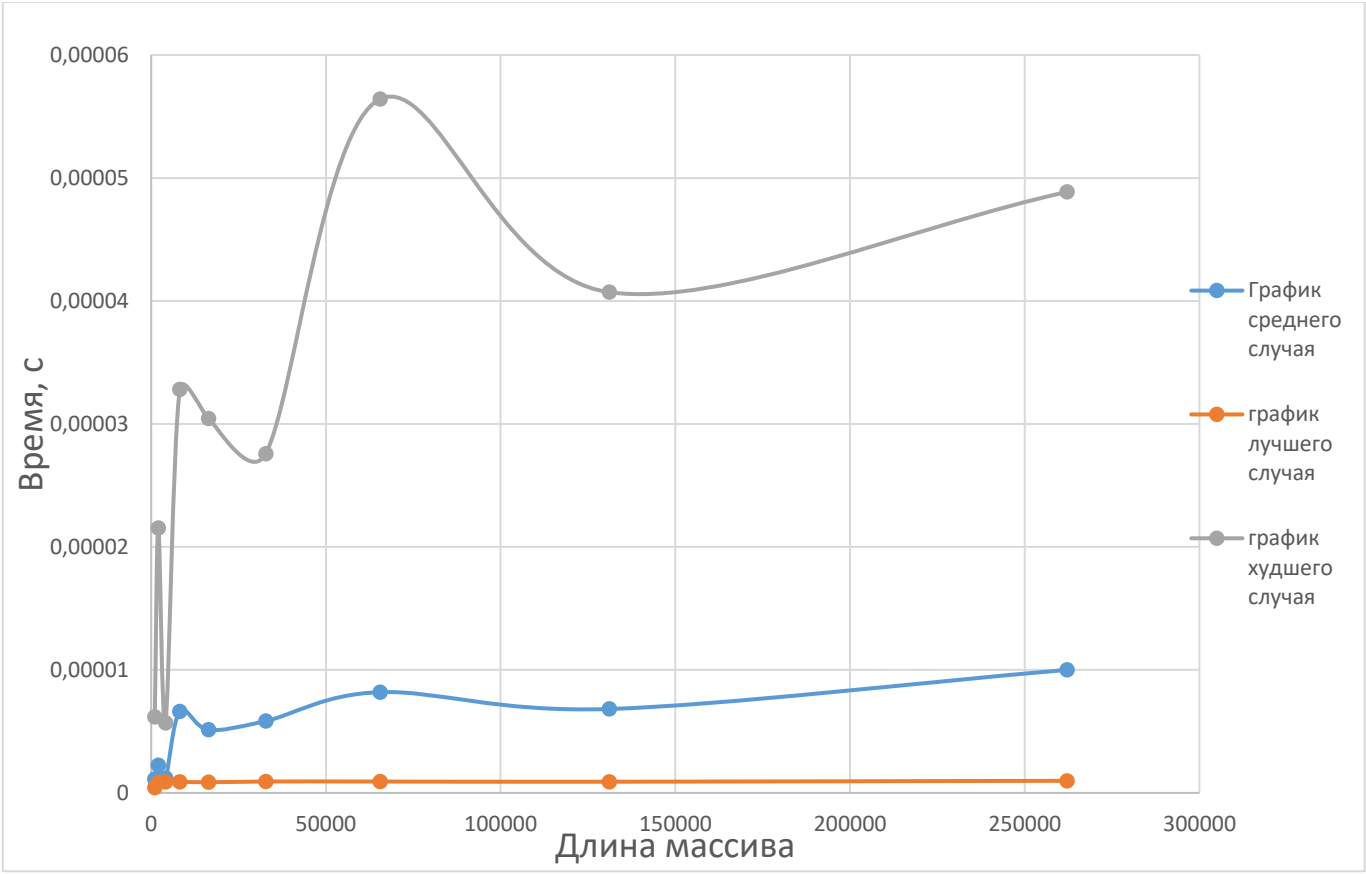

Рис.3. График времени, за которое выполняется вставка в сортированное Декартово дерево

По графикам видно, что в виду автобалансировки, дерево с отсортированными заранее значениями сильно не влияют на время процесса.

4) Замерил время вставки несортированного AVL-дерева

Таблица 3

Время вставки несортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000557	0.000014776	0.000001879
11	0.000000793	0.000005361	0.000001447
12	0.000000684	0.000002169	0.000001517
13	0.000000869	0.000002636	0.000001793
14	0.000001135	0.000014728	0.000002421
15	0.000001768	0.000012356	0.000002923
16	0.000001636	0.000005289	0.000003065
17	0.000001809	0.000004374	0.000003269
18	0.000002392	0.000021515	0.000004716

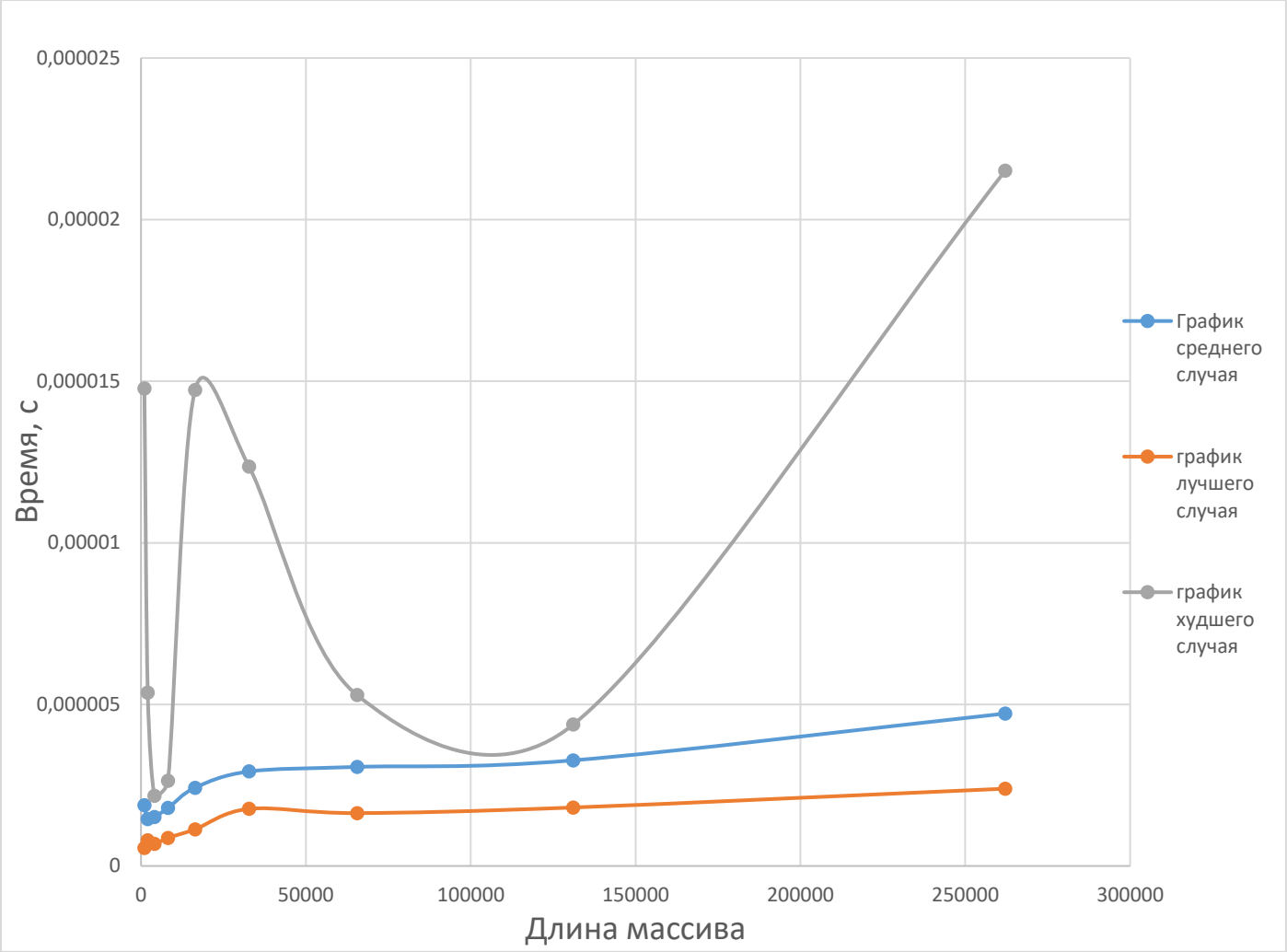

Рис.4. График времени, за которое выполняется вставка у несортированного AVL-дерева

5) Замерил время вставки сортированного AVL-дерева

Таблица 4

Время вставки сортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000357	0.000002488	0.000000502
11	0.000000383	0.000001092	0.000000437
12	0.000000386	0.000000634	0.000000443
13	0.000000421	0.000000586	0.000000483
14	0.000000472	0.000001600	0.000000553
15	0.000000504	0.000001116	0.000000581
16	0.000000528	0.000000749	0.000000588
17	0.000000559	0.000001571	0.000000626
18	0.000000595	0.000000869	0.000000639

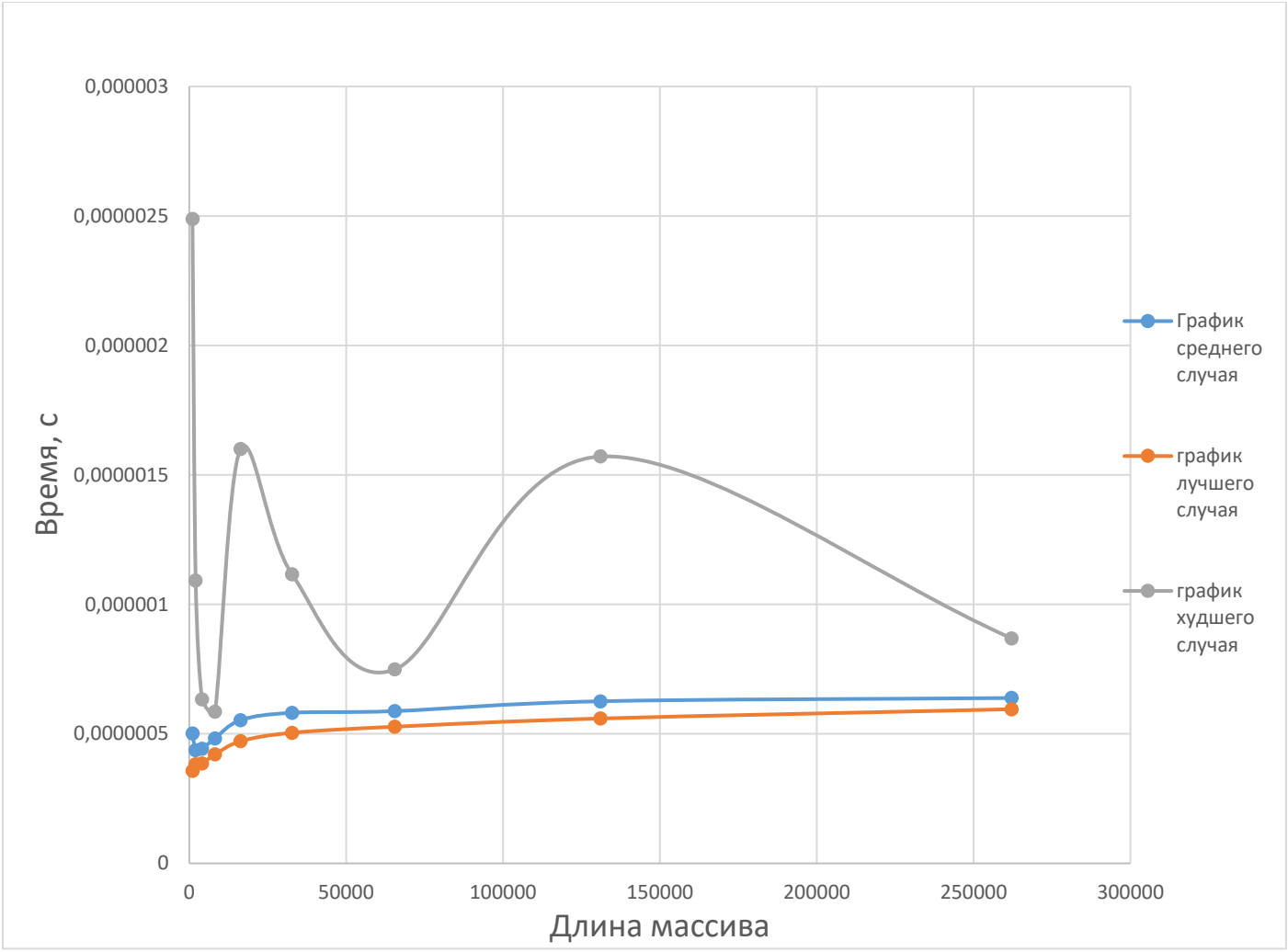

Рис.5. График времени, за которое выполняется вставка у сортированного AVL-дерева

По графикам видно, что характер описания вставки похожий(логарифмический) и по времени заранее отсортированное дерево работает иногда быстрее.

6) Замерил удаление у несортированного Бинарного дерева

Таблица 5

Время удаления одного элемента у несортированного Бинарного дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000772	0.000007462	0.000001891
11	0.000000848	0.000004039	0.000001724
12	0.000000958	0.000009425	0.000002676
13	0.000001074	0.000011808	0.000003138
14	0.000001219	0.000014032	0.000003755
15	0.000001403	0.000028774	0.000006378
16	0.000001626	0.000044997	0.000007233
17	0.000001717	0.000031700	0.000008181
18	0.000001955	0.000108497	0.000013127

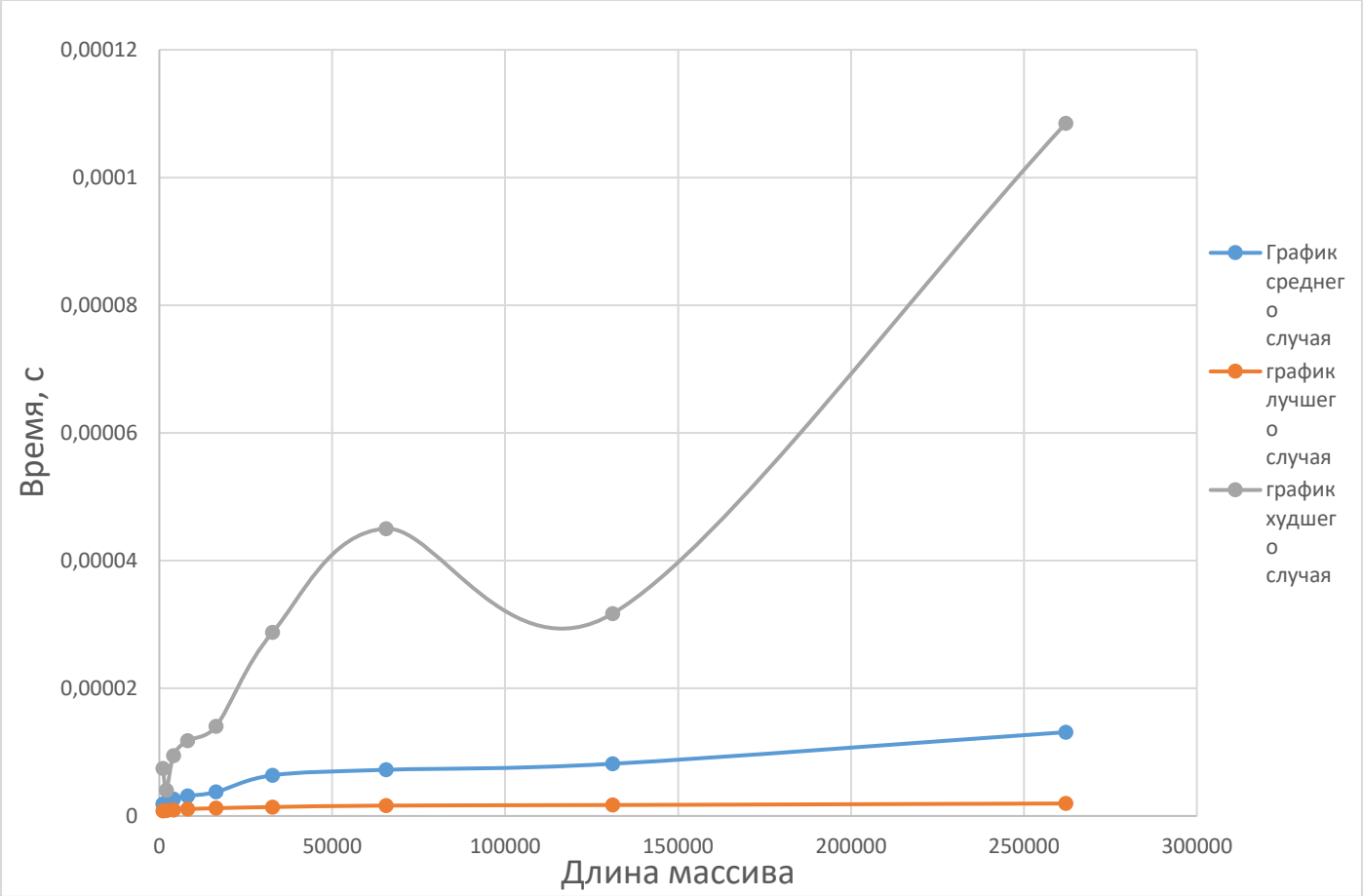

Рис.6. График времени, за которое удаляется узел в несортированном Декартовом дереве

7) Замерил удаления у сортированного Декартова дерева

Таблица 6

Время удаления одного элемента у сортированного Декартова дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000716	0.000010356	0.000001901
11	0.000000865	0.000004235	0.000001911
12	0.000000936	0.000006988	0.000002427
13	0.000001087	0.000015275	0.000003312
14	0.000001248	0.000044388	0.000006086
15	0.000001500	0.000046128	0.000006653
16	0.000001687	0.000028113	0.000007805
17	0.000001826	0.000059696	0.000008917
18	0.000002121	0.000070485	0.000011740

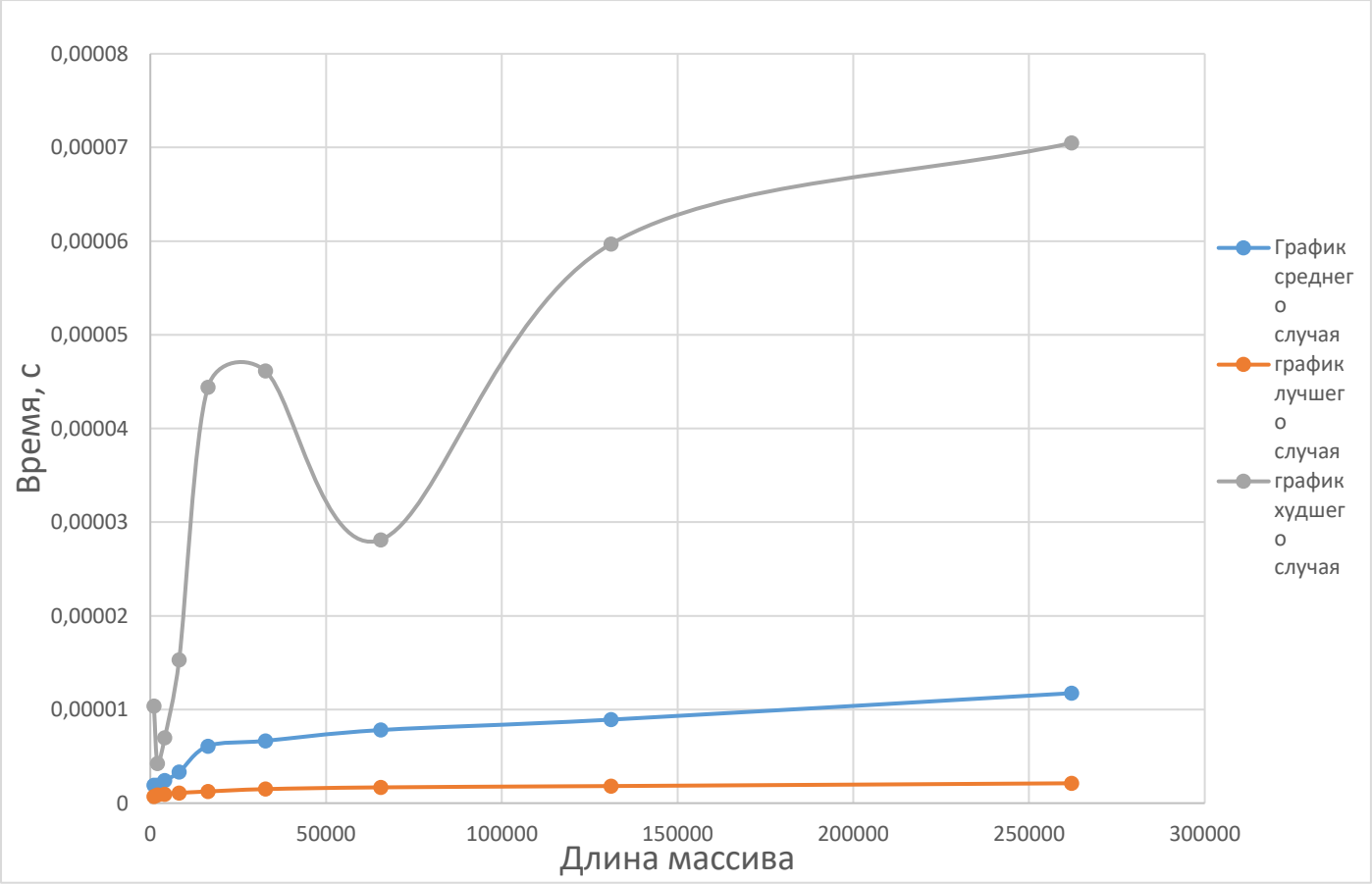

Рис.7. График времени, за которое ищется узел в сортированном Бинарном дереве

По графикам видно, что характер описания удаления логарифмический и оба дерева удаляют за примерно равное время.

8) Замерил удаление у несортированного AVL-дерева

Таблица 7

Время удаления одного элемента у несортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000649	0.000007449	0.000001596
11	0.000000791	0.000003012	0.000001298
12	0.000000765	0.000003066	0.000001458
13	0.000000845	0.000002629	0.000001831
14	0.000001180	0.000003317	0.000002209
15	0.000001555	0.000004435	0.000002766
16	0.000001716	0.000006542	0.000003322
17	0.000001903	0.000004349	0.000003291
18	0.000002474	0.000005524	0.000003839

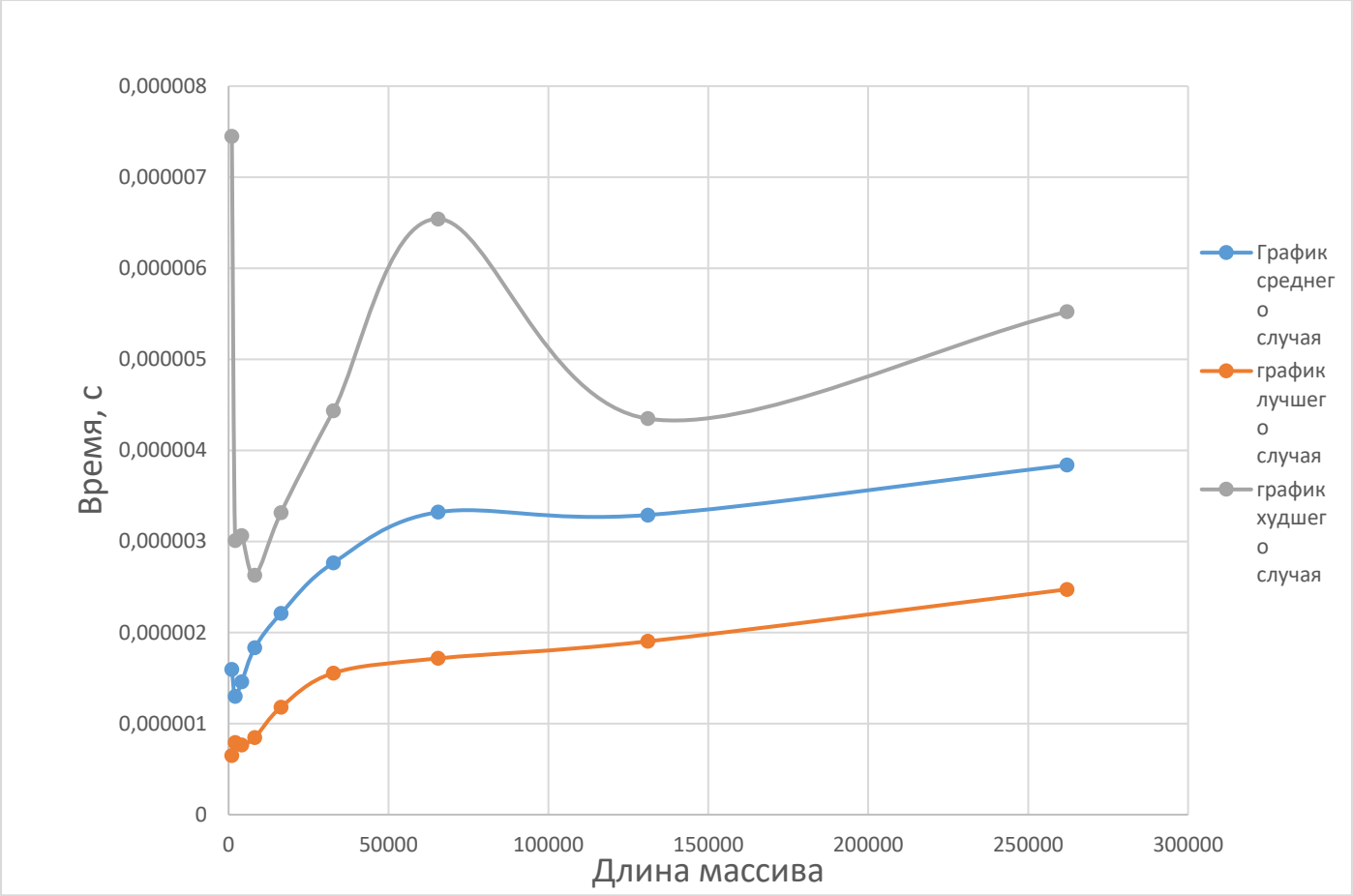

Рис.8. График времени, за которое удаляется узел в несортированном AVL-дереве

9) Замерил удаление у сортированного AVL-дерева

Таблица 8

Время удаления одного элемента у сортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000469	0.000016665	0.000001136
11	0.000000534	0.000001630	0.000000670
12	0.000000596	0.000003430	0.000000881
13	0.000000689	0.000001554	0.000000992
14	0.000000891	0.000001660	0.000001169
15	0.000001135	0.000002375	0.000001484
16	0.000001263	0.000003663	0.000001703
17	0.000001491	0.000002029	0.000001647
18	0.000001690	0.000003563	0.000001964

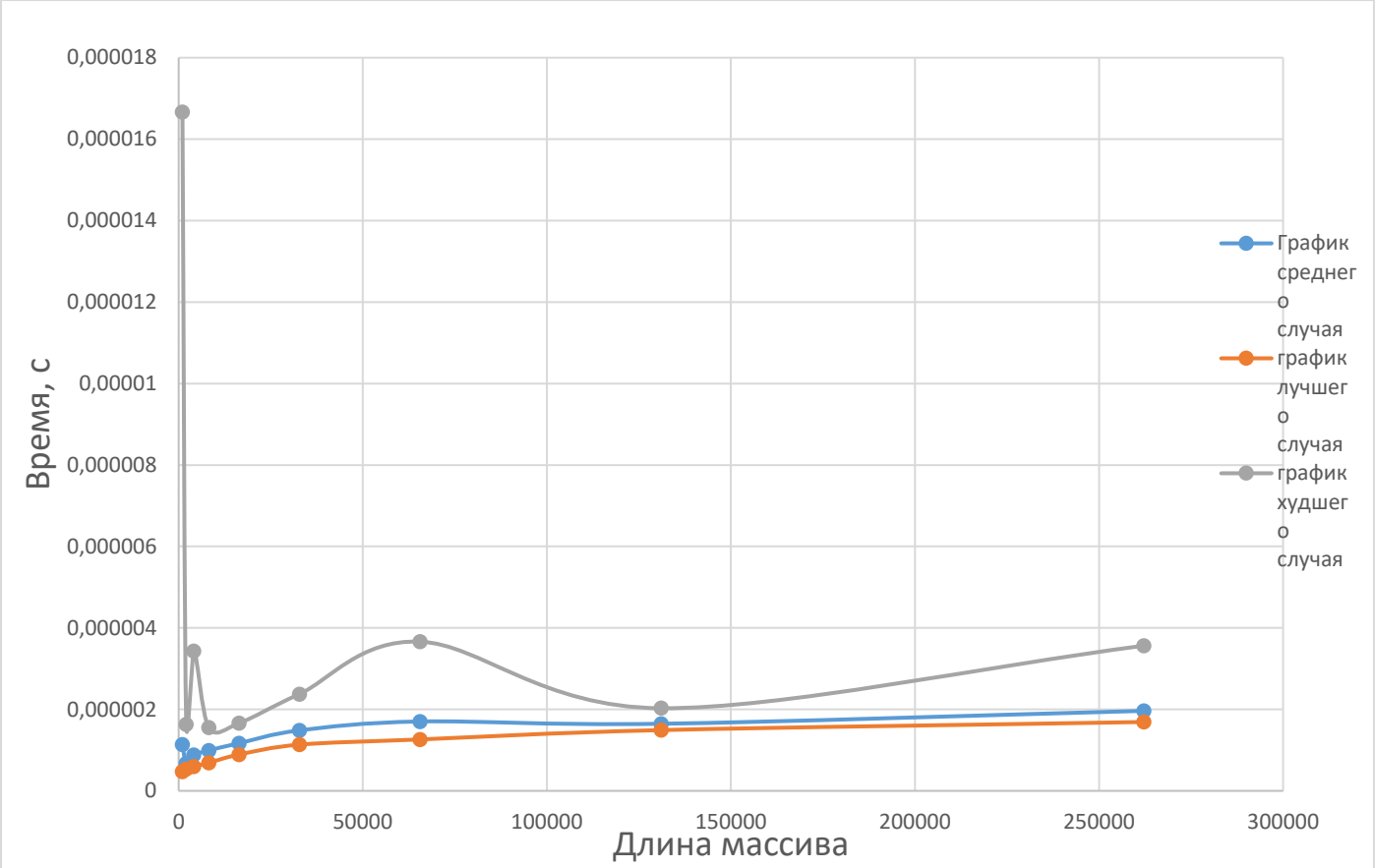

Рис.9. График удаления, за которое удаляется узел в сортированном AVL-дереве

По графикам видно, что удаление у AVL-дерева описывается логарифмически не сильно зависит от того, был ли контейнер отсортирован заранее.

10) Замерил поиск у несортированного Декартова дерева

Таблица 9

Время поиска одного элемента у несортированного Декартова-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000171	0.000000545	0.000000224
11	0.000000186	0.000000312	0.000000233
12	0.000000225	0.000000381	0.000000280
13	0.000000282	0.000000417	0.000000320
14	0.000000324	0.000000899	0.000000400
15	0.000000444	0.000000619	0.000000492
16	0.000000539	0.000001537	0.000000689
17	0.000000639	0.000001187	0.000000717
18	0.000000741	0.000001620	0.000000877

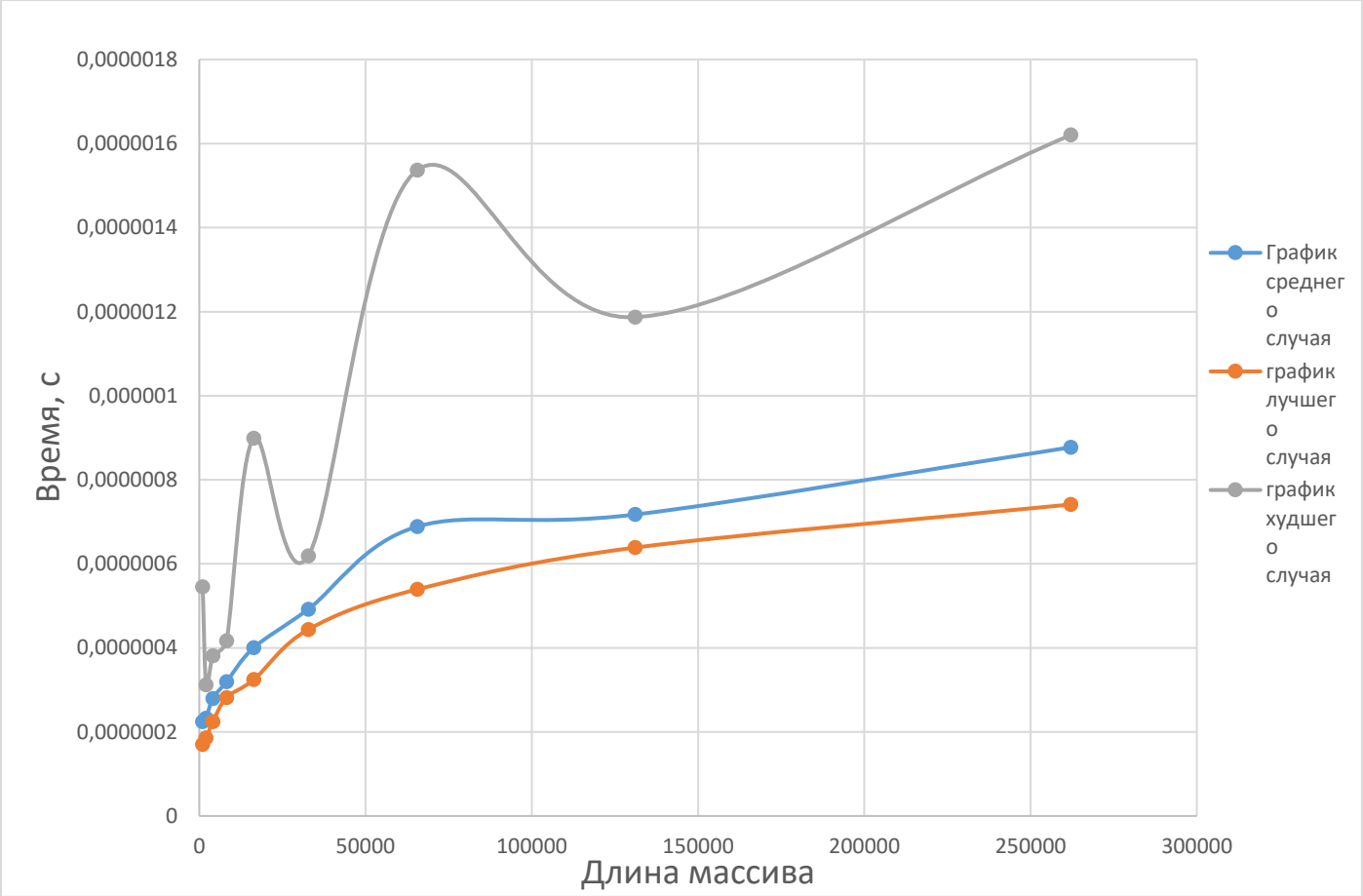

Рис.10. График времени, за которое удаляется узел в несортированном Декартовом дереве

11) Замерил поиска у сортированного Декартова дерева

Таблица 10

Время поиска одного элемента у сортированного Декартова дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000144	0.000003326	0.000000298
11	0.000000173	0.000000718	0.000000217
12	0.000000205	0.000000469	0.000000244
13	0.000000241	0.000000504	0.000000302
14	0.000000322	0.000000805	0.000000394
15	0.000000453	0.000001854	0.000000596
16	0.000000557	0.000003154	0.000000759
17	0.000000651	0.000001069	0.000000740
18	0.000000787	0.000001669	0.000000922

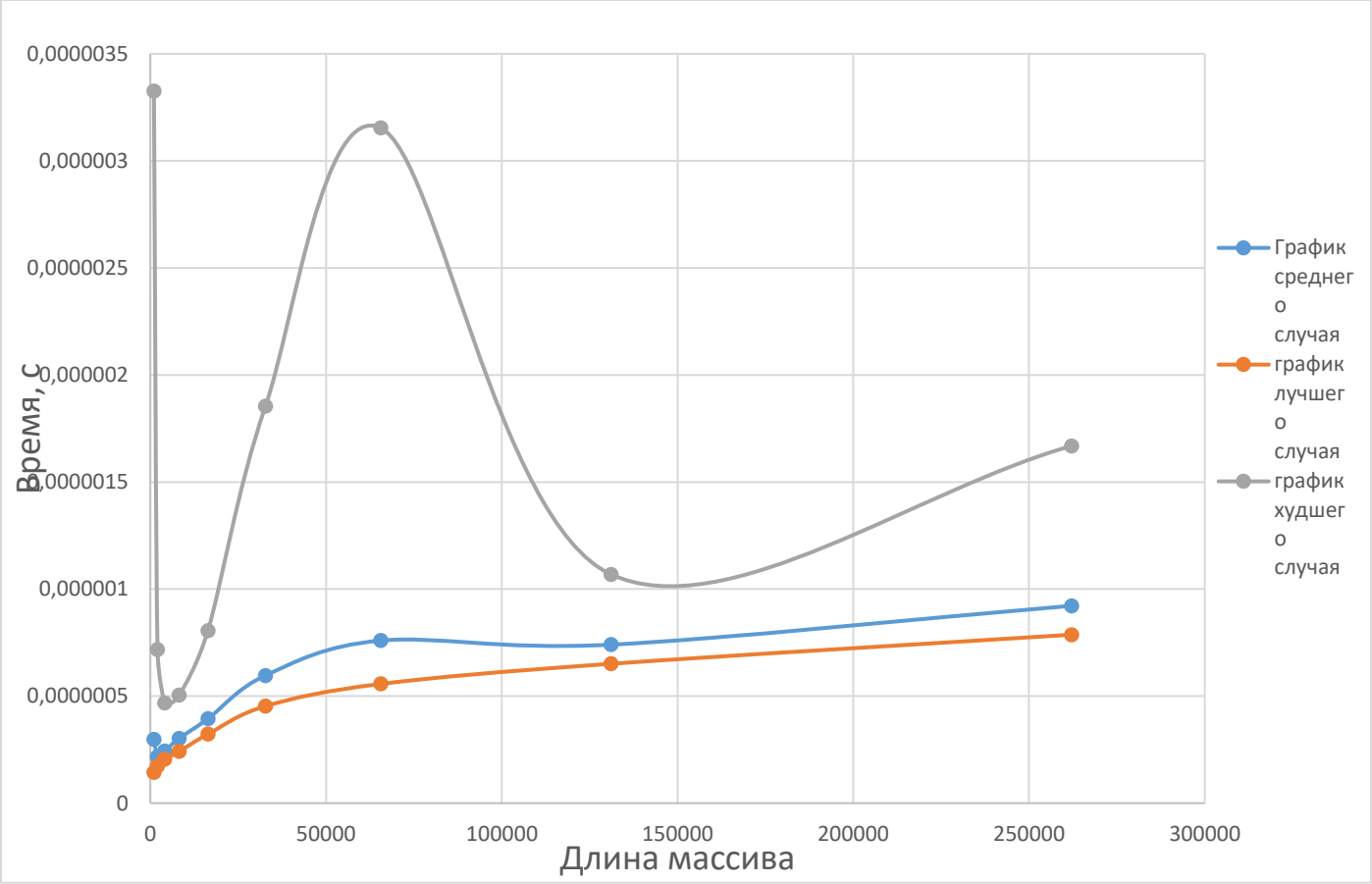

Рис.11. График времени, за которое ищется узел в сортированном Бинарном дереве

По графикам видно, что поиск имеет логарифмический характер и не теряет сильно во времени на заранее отсортированных данных.

12) Замерил поиск у несортированного AVL-дерева

Таблица 11

Время поиск одного элемента у несортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000190	0.000002524	0.000000587
11	0.000000256	0.000001391	0.000000501
12	0.000000192	0.000001131	0.000000585
13	0.000000223	0.000001595	0.000000736
14	0.000000334	0.000001178	0.000000887
15	0.000000513	0.000001905	0.000001218
16	0.000000844	0.000005710	0.000001684
17	0.000000747	0.000002160	0.000001492
18	0.000001020	0.000002474	0.000001823

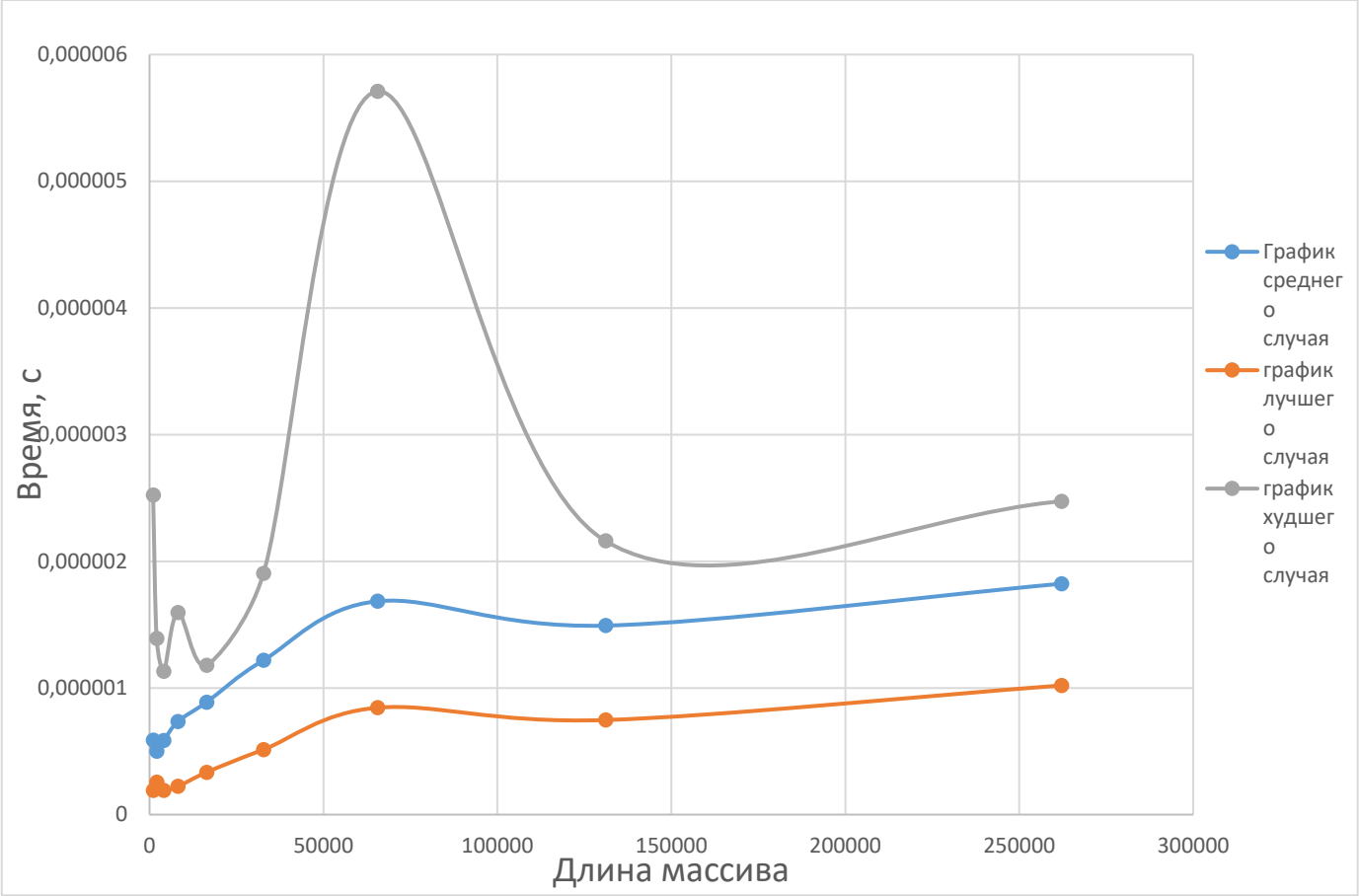

Рис.12. График времени, за которое ищется узел в несортированном AVL-дереве

13) Замерил поиск у сортированного AVL-дерева

Таблица 12

Время поиска одного элемента у сортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000000126	0.000000771	0.000000165
11	0.000000159	0.000000711	0.000000209
12	0.000000177	0.000001640	0.000000259
13	0.000000208	0.000000485	0.000000290
14	0.000000268	0.000000856	0.000000365
15	0.000000393	0.000002687	0.000000587
16	0.000000461	0.000003908	0.000000767
17	0.000000586	0.000001100	0.000000682
18	0.000000717	0.000001941	0.000000869

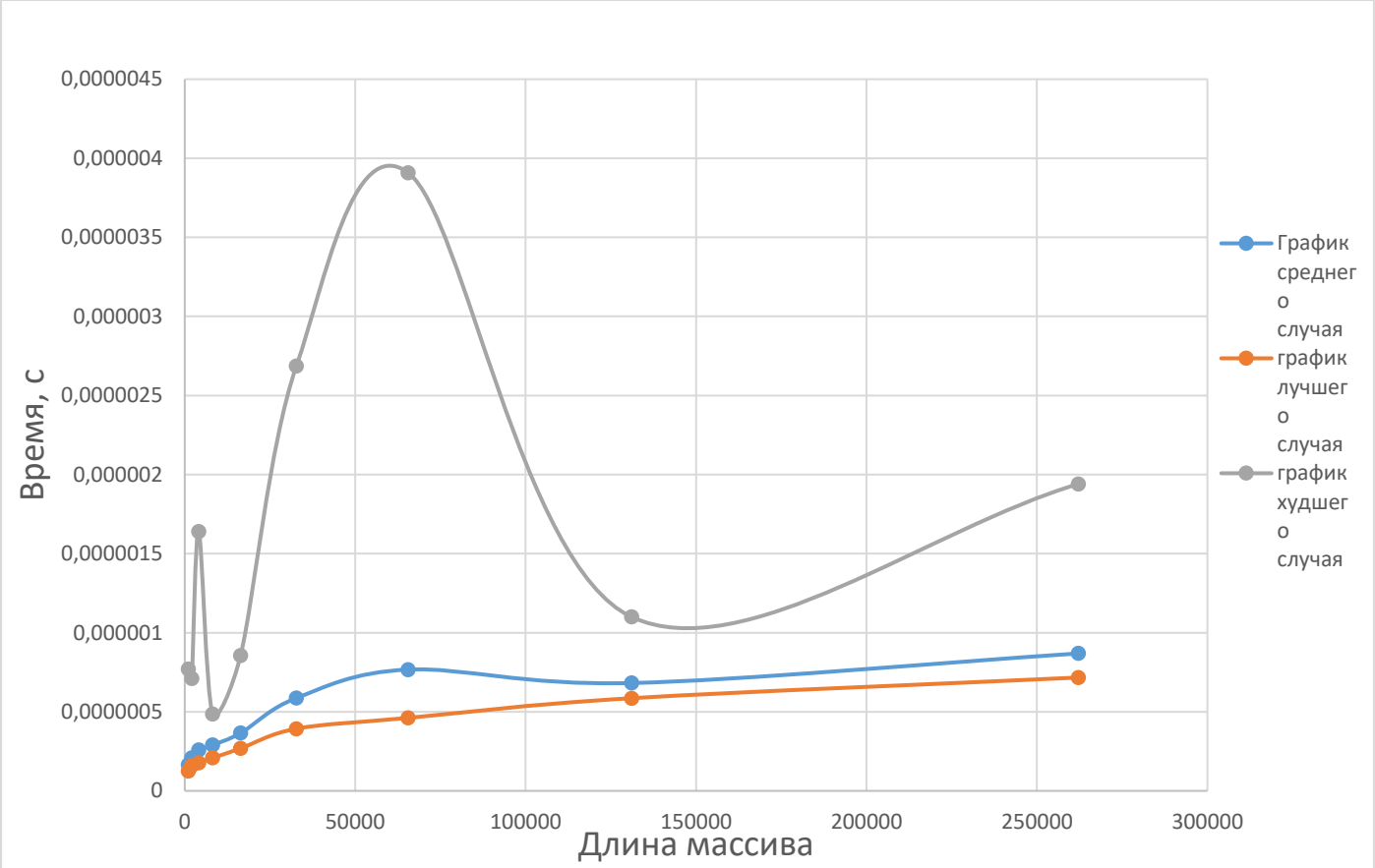

Рис.13. График времени, за которое ищется узел в сортированном AVL-дереве

По графикам видно, что поиск имеет логарифмический характер и не теряет сильно во времени на заранее отсортированных данных.

14)Макс. Высота несортированного Декартова дерева

Таблица 13

Макс. Высота несортированного Декартова дерева на каждой из серии тестов

2^	Макс.Высота
10	28
11	30
12	37
13	37
14	41
15	42
16	43
17	45
18	53

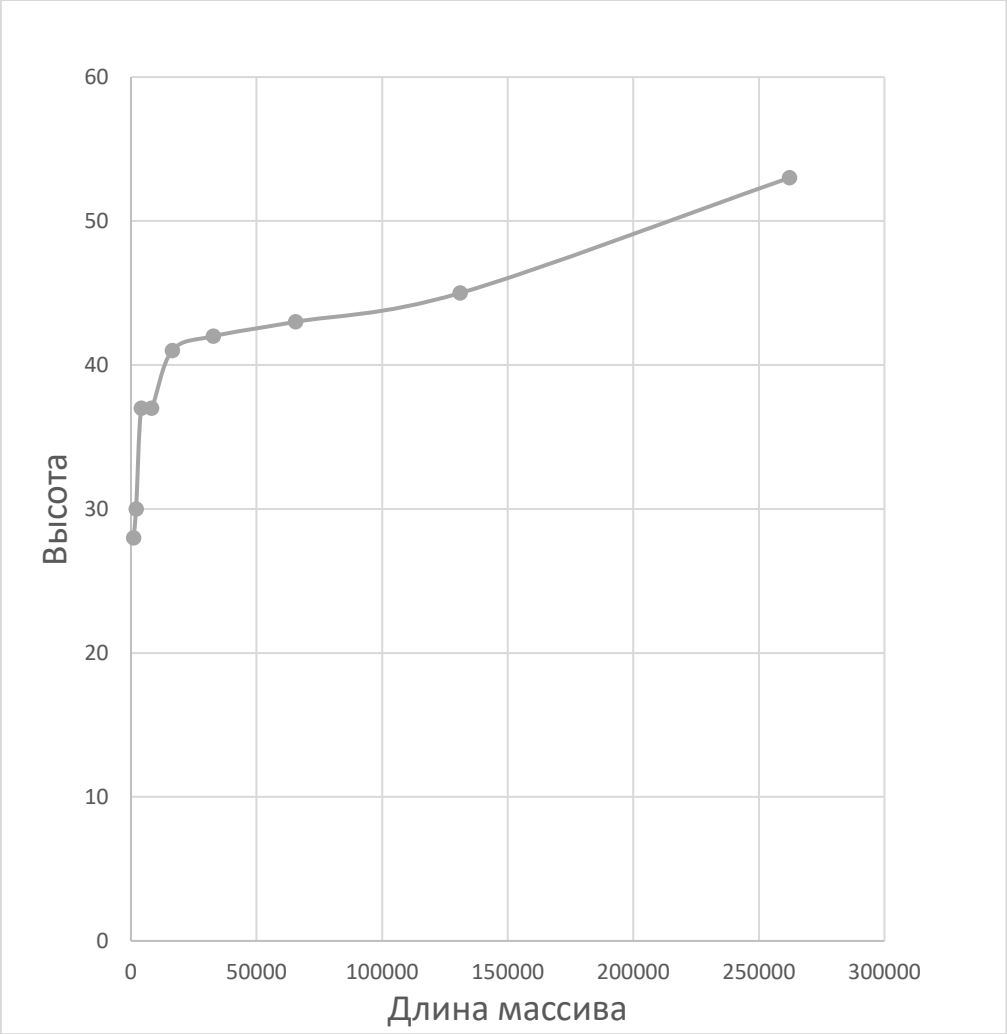

Рис.14. График изменения макс. Высоты Декартово дерева с увеличением длины массива

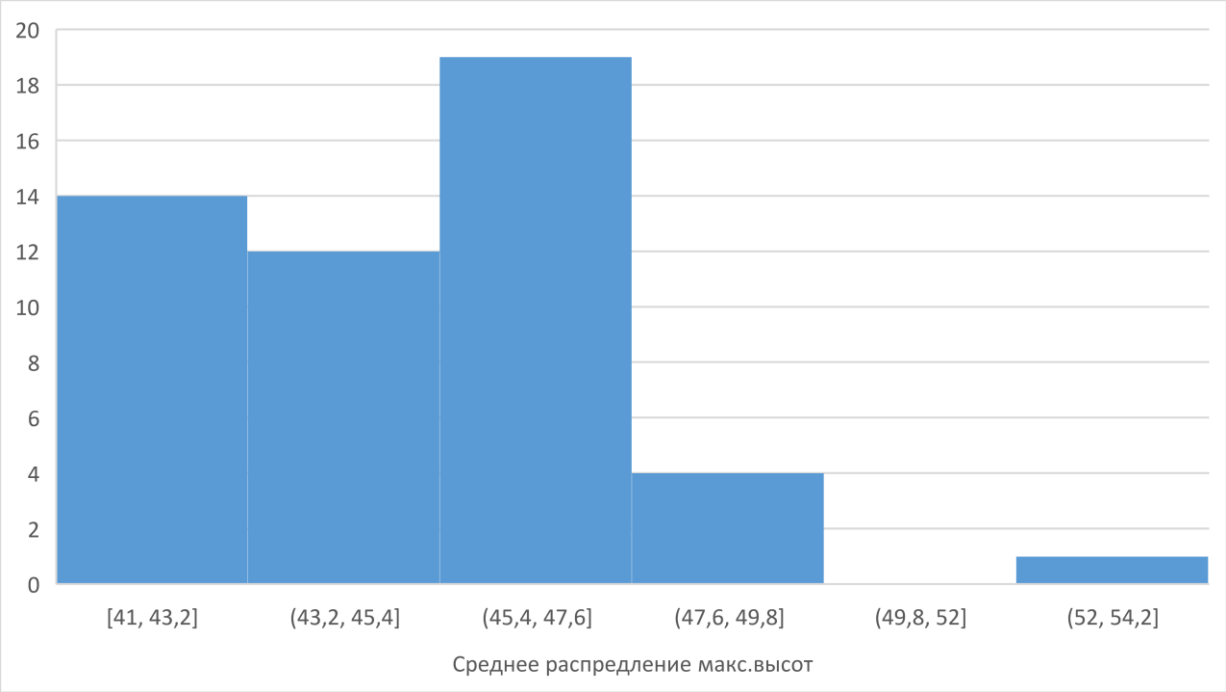

Рис.15. Гистограмма среднего распределения макс. Высот Декартово дерева на последней серии тестов

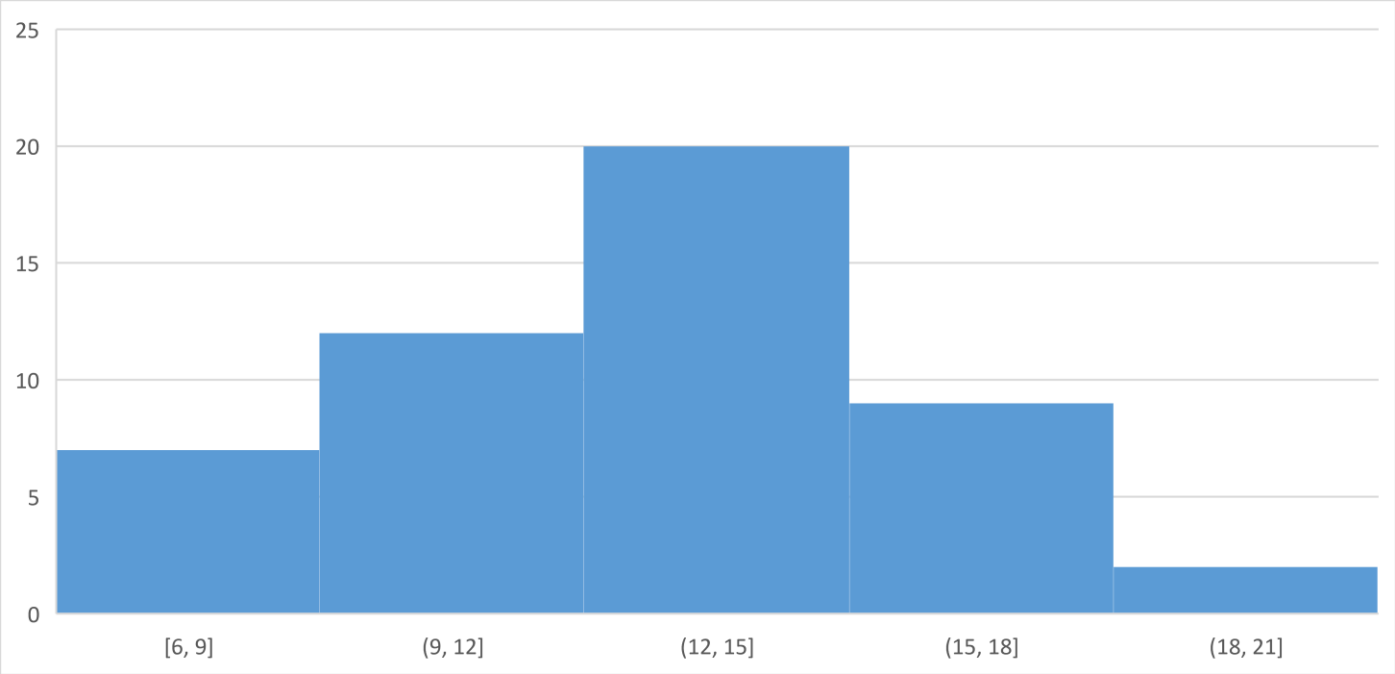

Рис.16. Гистограмма среднего распределения макс. Высот всех веток дерева на последней серии тестов

15)Макс. Высота сортированного Декартова дерева

Таблица 14

Макс. Высота сортированного Декартова дерева на каждой из серии тестов

2^	Макс.Высота
10	11
11	11
12	15
13	15
14	17
15	19
16	18
17	19
18	21

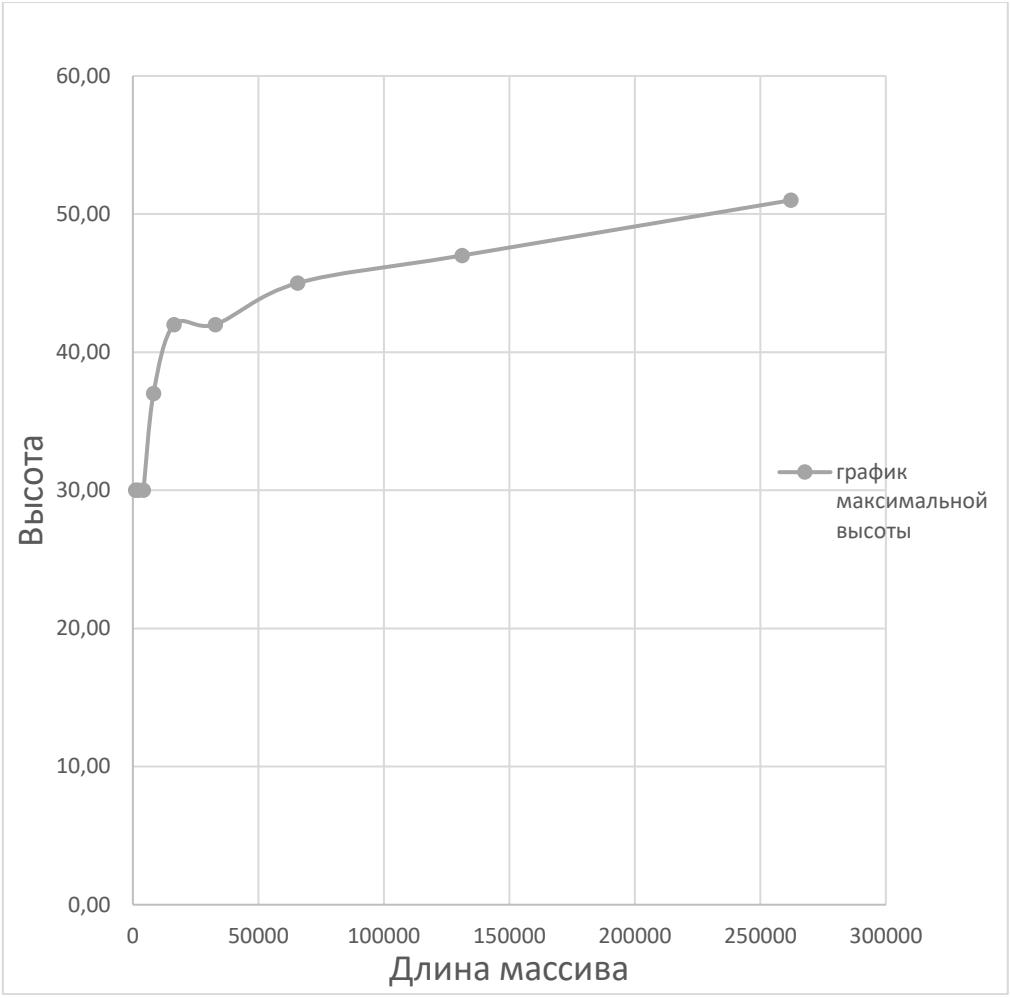

Рис.17. График изменения макс. Высоты сортированного Декартово дерева с увеличением длины массива

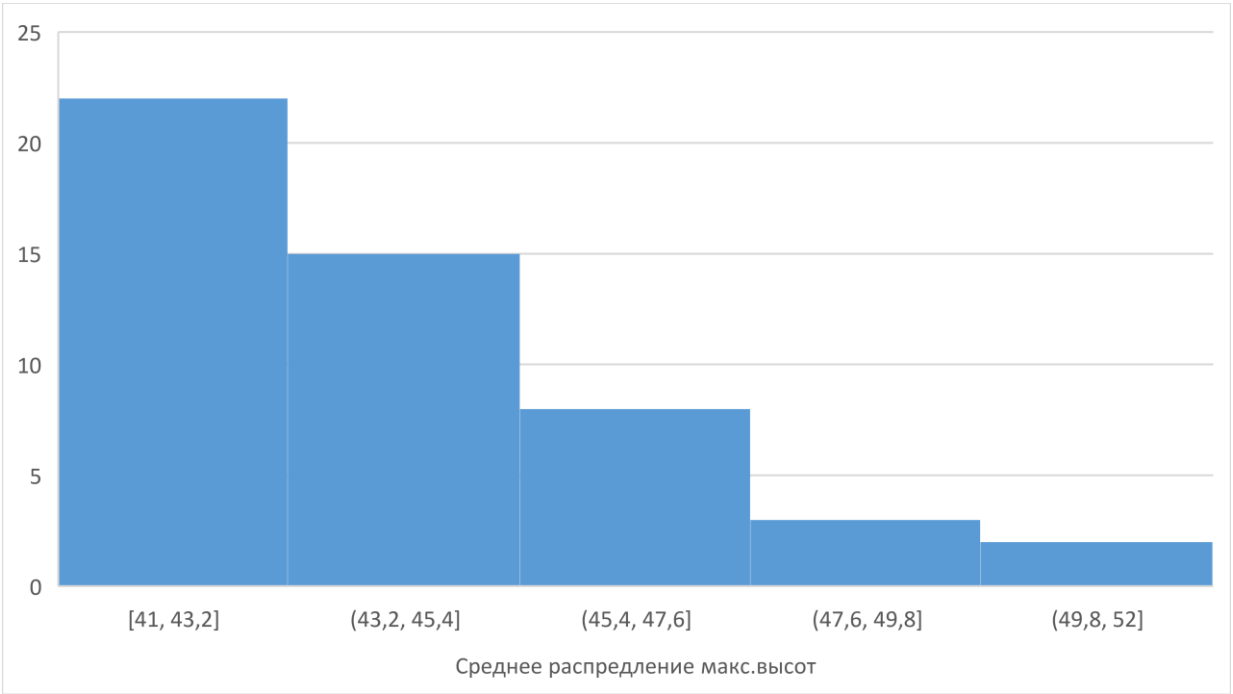

Рис.18. Гистограмма среднего распределения макс. Высот сорт. Декартово дерева на последней серии тестов

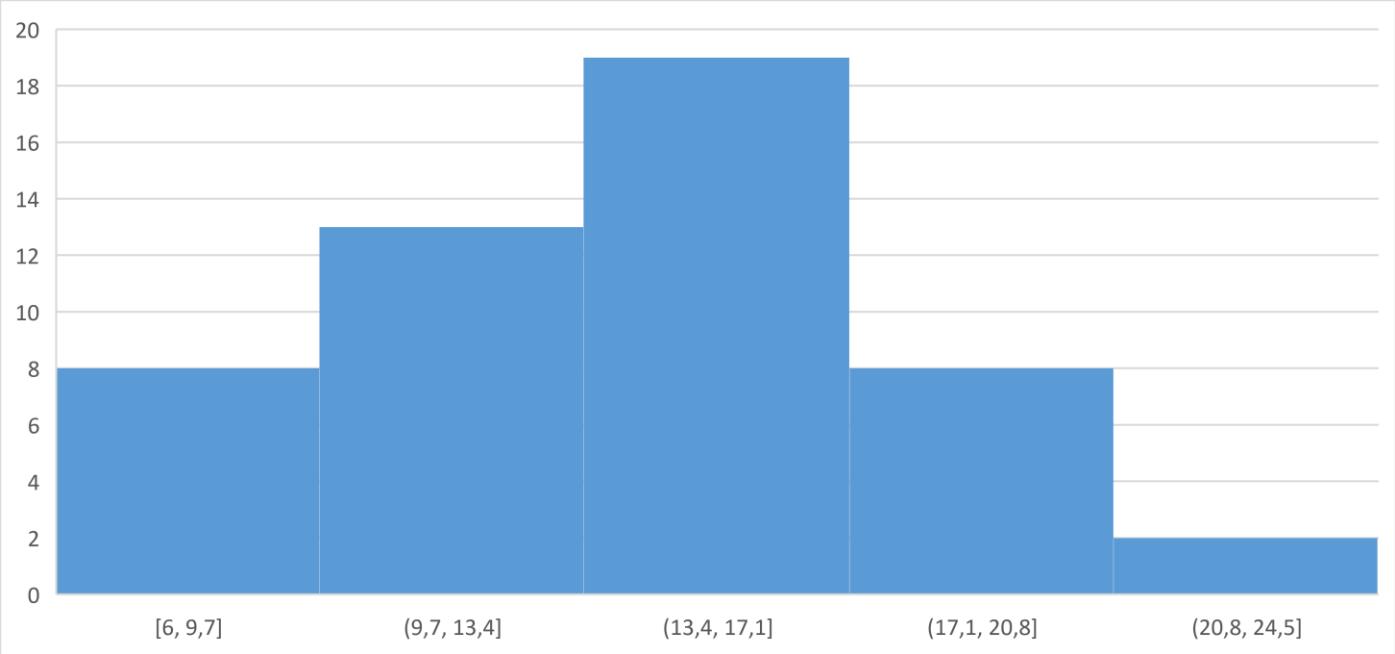

Рис.19. Гистограмма среднего распределения макс. Высот всех веток сорт. Декартова дерева на последней серии тестов

16)Макс. Высота несортированного AVL-дерева

Таблица 15

Макс. Высота несортированного AVL-дерева на каждой из серии тестов

2^	Макс.Высота
10	19
11	20
12	21
13	22
14	24
15	25
16	26
17	27
18	28

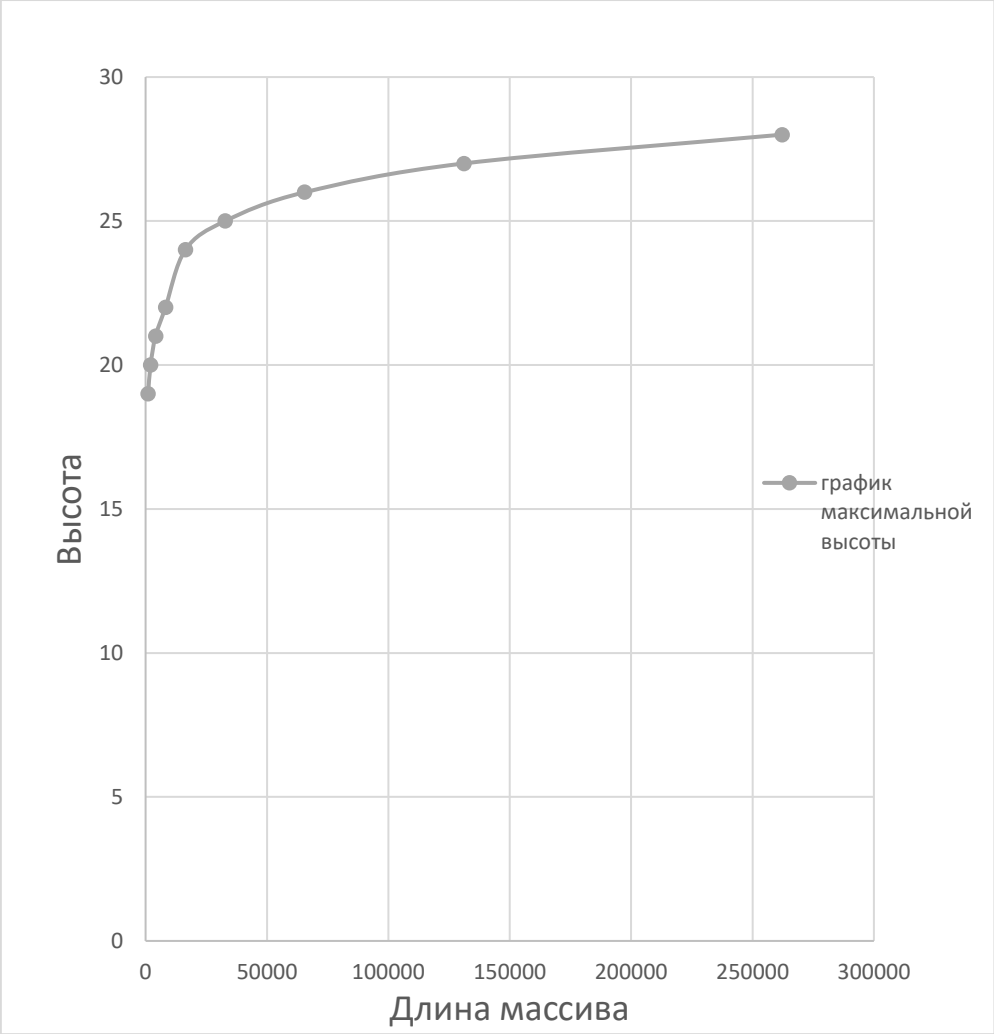

Рис.20. График изменения макс. Высоты несортированного AVL-дерева с увеличением длины массива

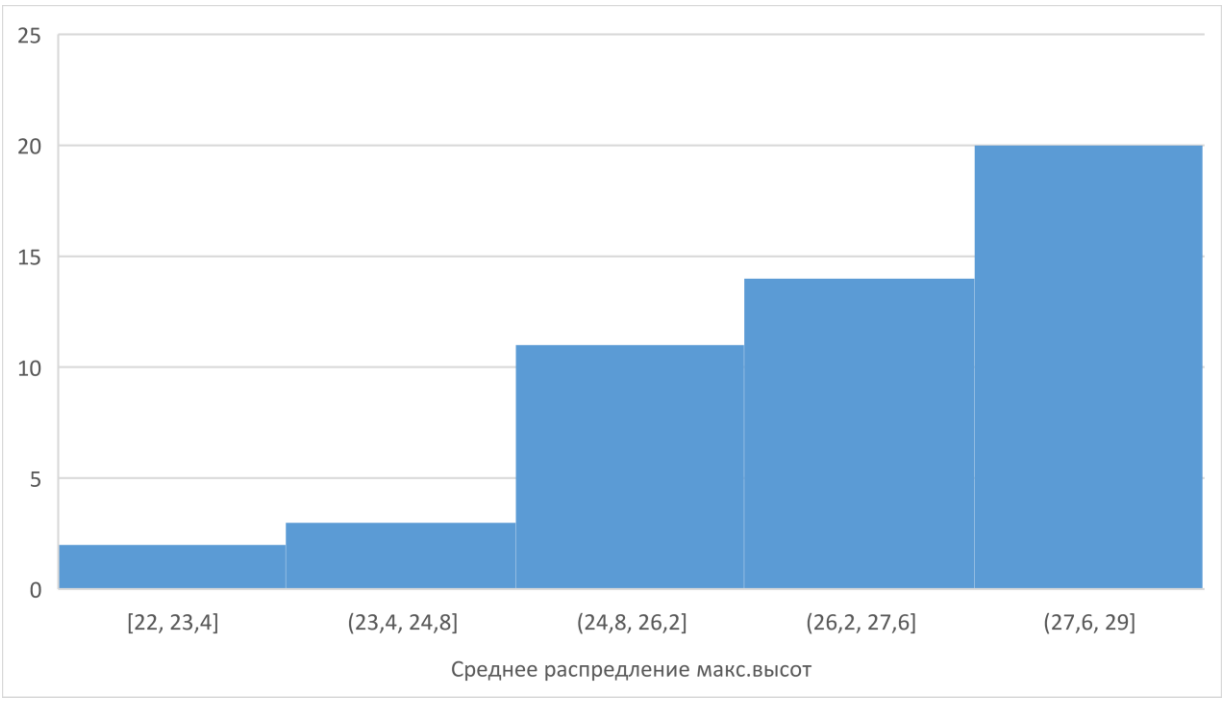

Рис.21. Гистограмма среднего распределения макс. Высот несорт. AVL -дерева на последней серии тестов

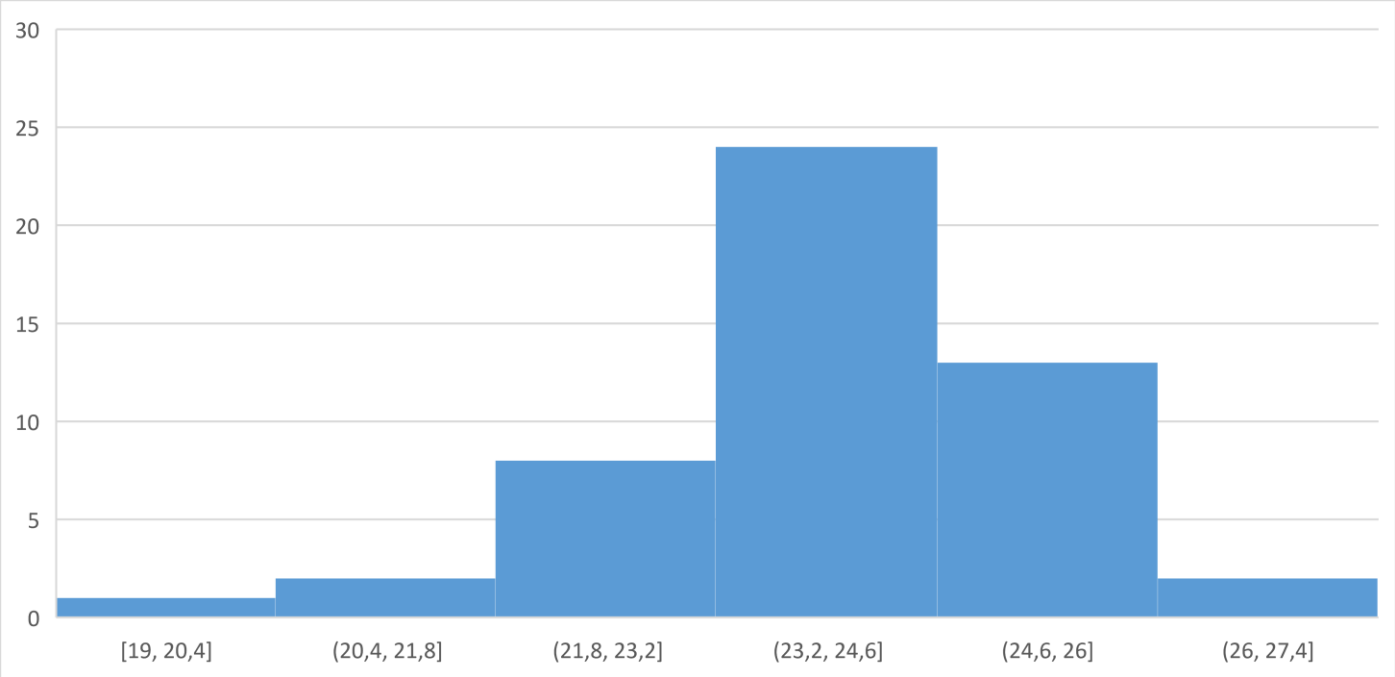

Рис.22. Гистограмма среднего распределения макс. Высот всех веток несорт. AVL -дерева на последней серии тестов

17)Макс. Высота сортированного AVL-дерева

Таблица 16

Макс. Высота сортированного AVL-дерева на каждой из серии тестов

2^	Макс.Высота
10	19
11	20
12	21
13	22
14	24
15	25
16	26
17	27
18	28

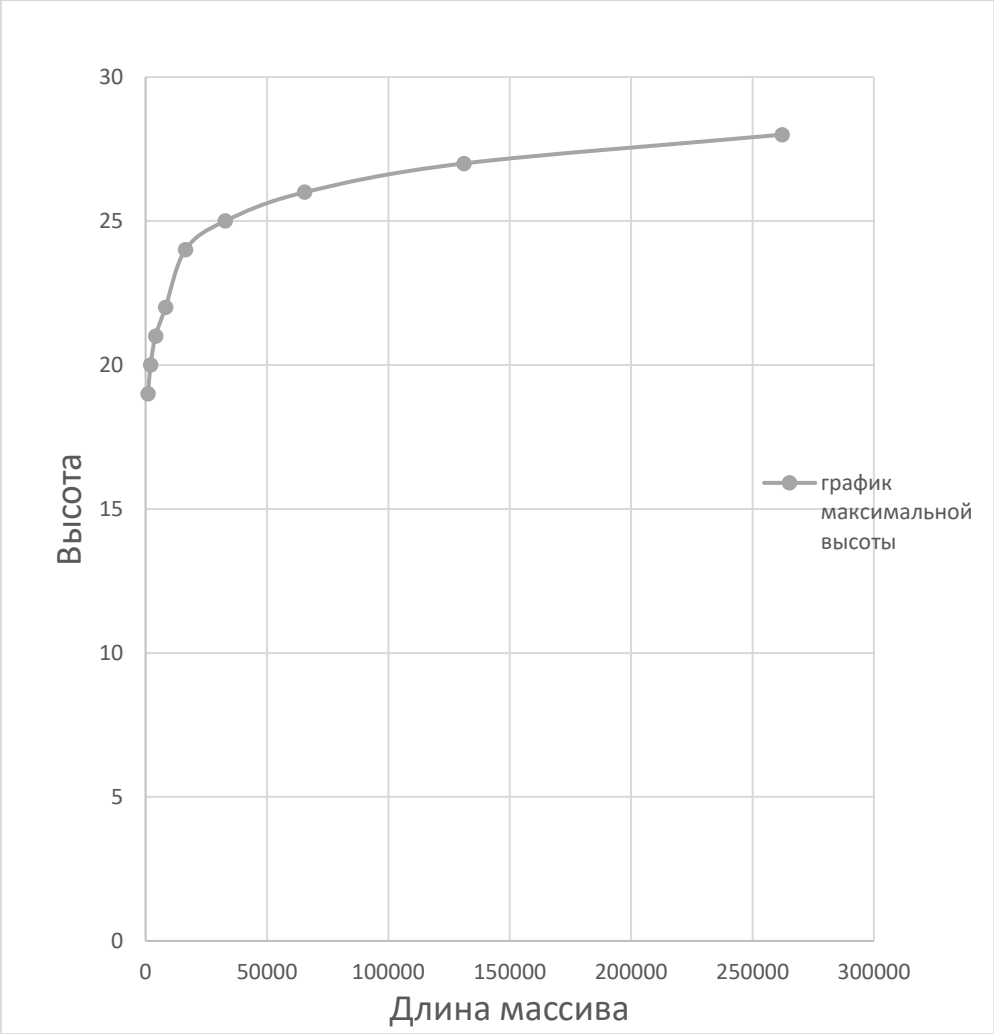

Рис.23. График изменения макс. Высоты сортированного AVL-дерева с увеличением длины массива

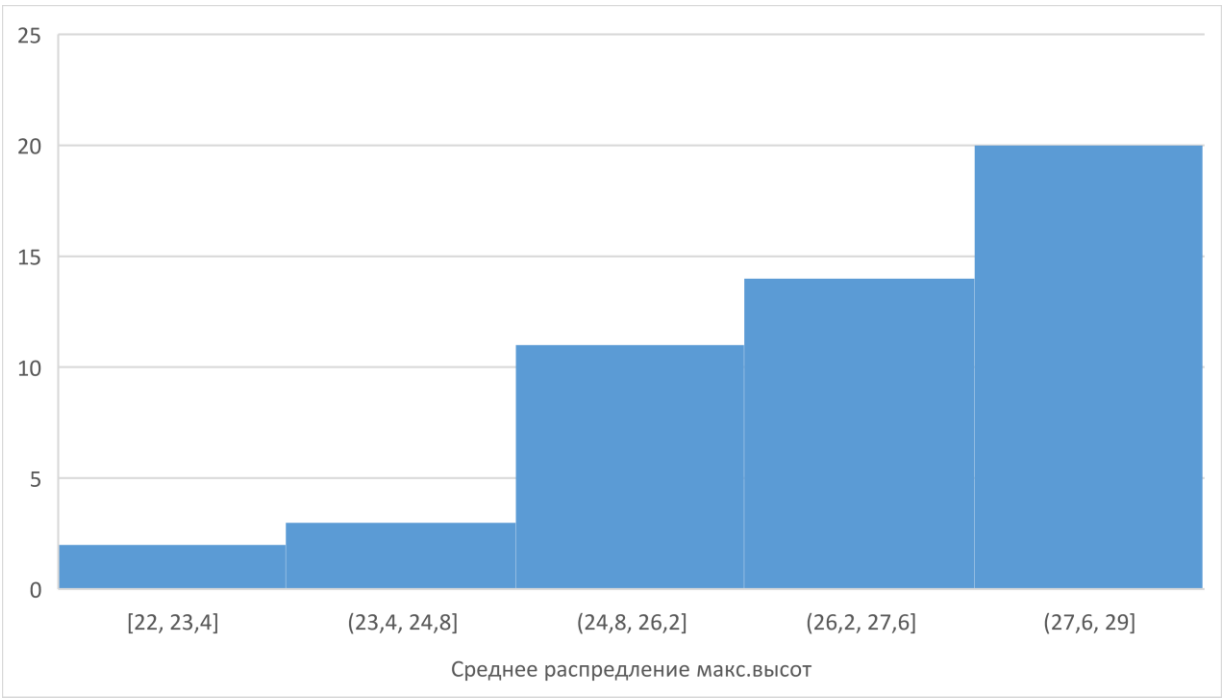

Рис.24. Гистограмма среднего распределения макс. Высот сорт. AVL -дерева на последней серии тестов

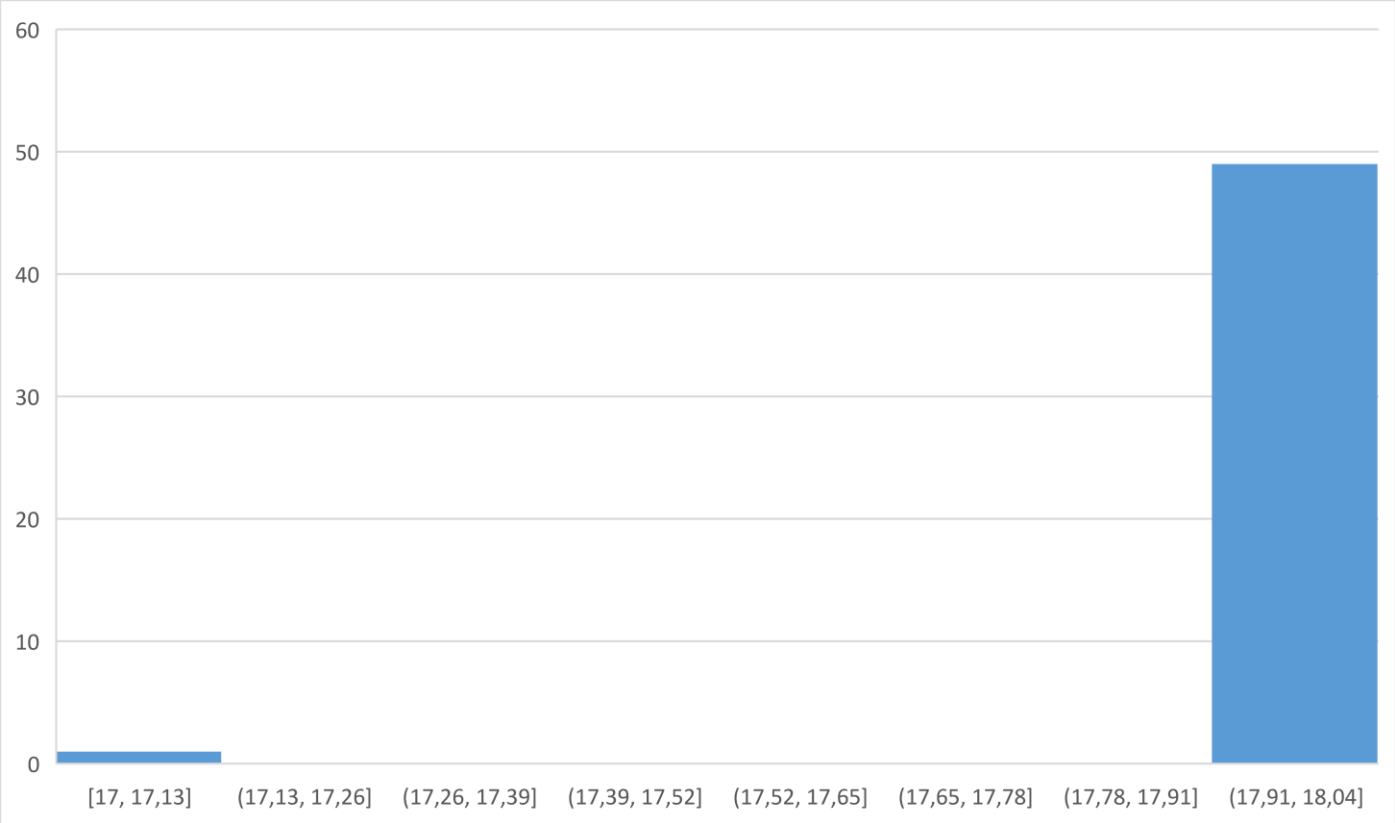

Рис.25. Гистограмма среднего распределения макс. Высот всех веток сорт. AVL -дерева на последней серии тестов

На всех графиках с зависимостью от длины массива можно было увидеть, что с возрастанием количества элементов логарифмически растет и время, за которое будут выполняться операции у AVL и Декартова дерева.

На графиках с распределением высот видно, что у AVL дерева при перебалансировке в среднем высота меньше у дерева, чем у декартово, что и дает константную прибавку к скорости для операций AVL-дереву.

18)Графики сравнения работы операции вставки:

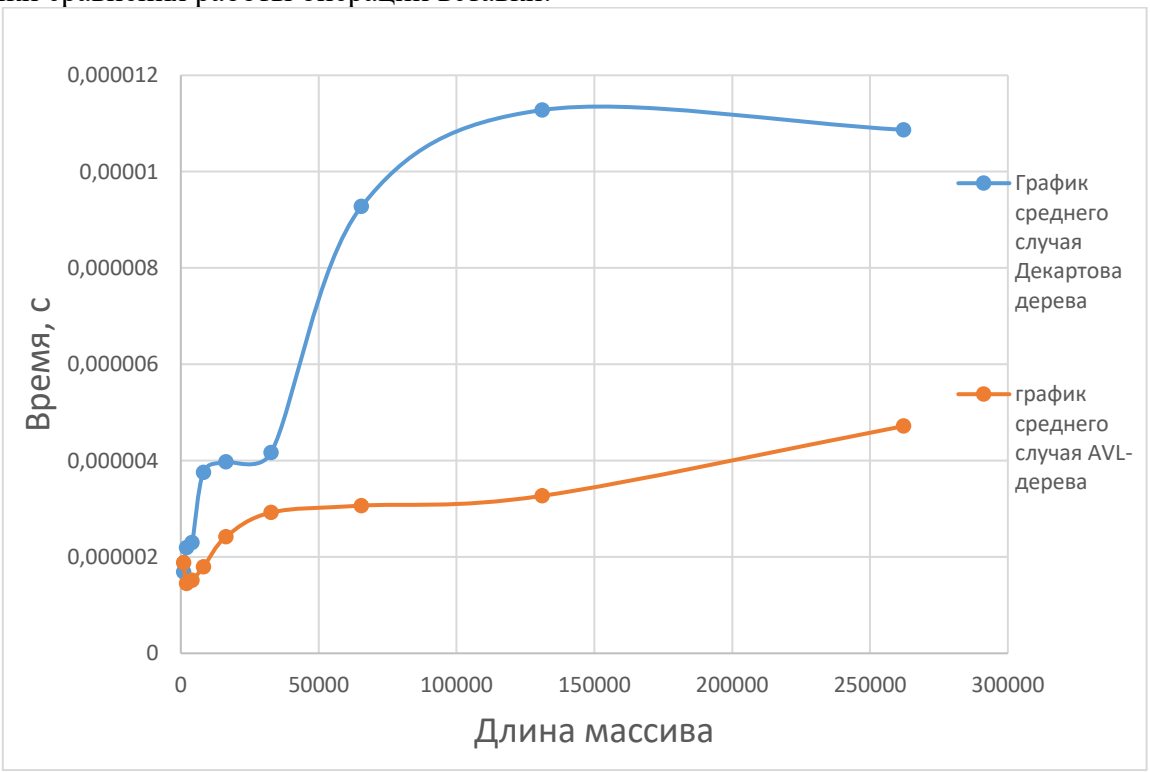

Рис.26. Графики сравнения работы операции вставки AVL и Декартова дерева

19)Графики сравнения работы операции удаления:

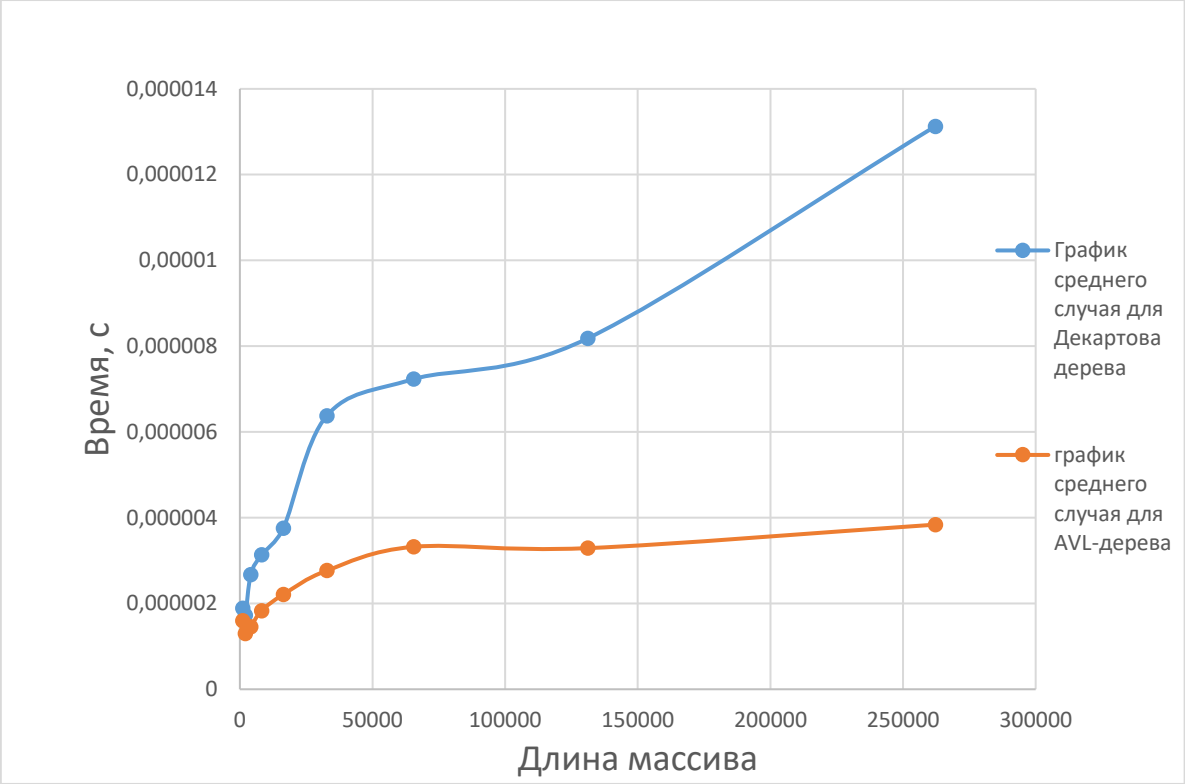

Рис.27. Графики сравнения работы операции удаления AVL и Декартова дерева

20)Графики сравнения работы операции поиска:

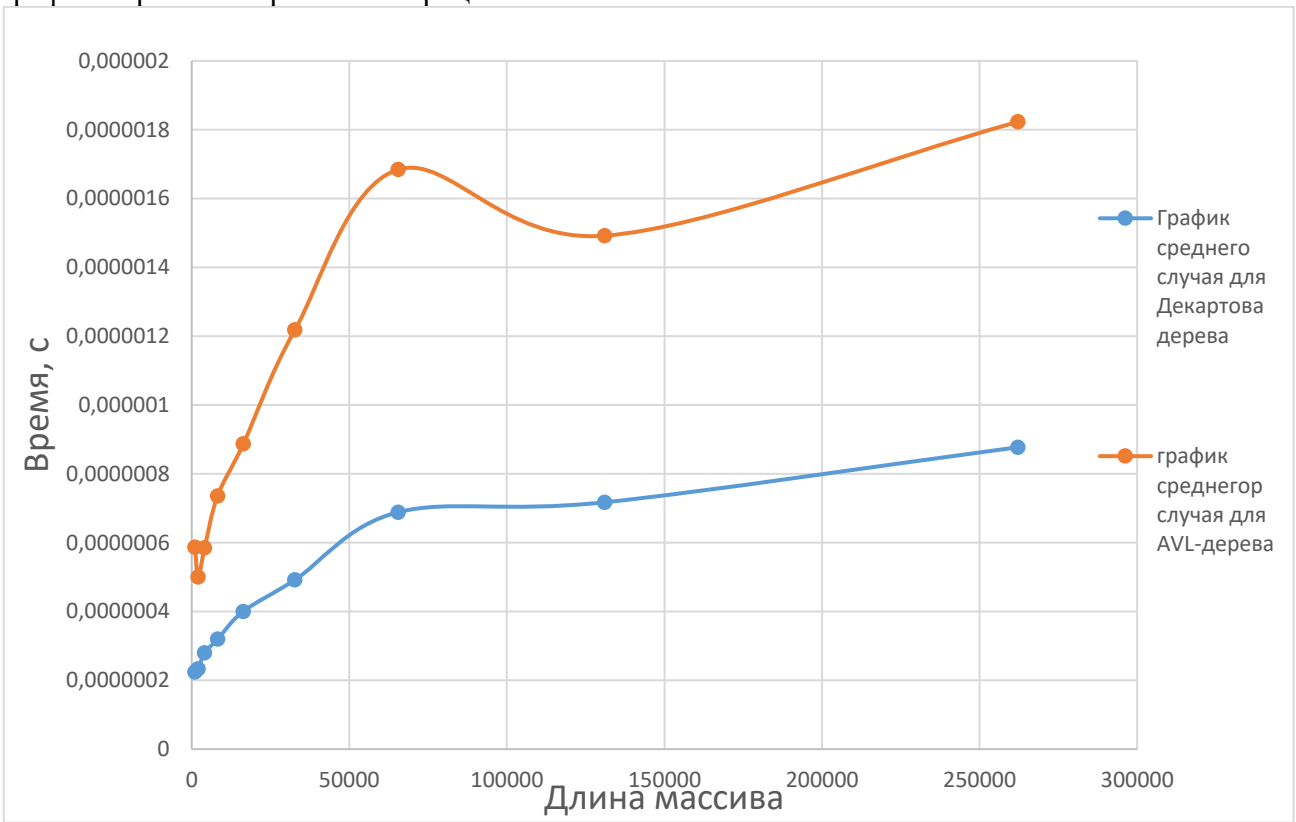

Рис.28. Графики сравнения работы операции удаления AVL и Декартова дерева

Операция вставки и удаления выполняются быстрее AVL-деревом, а поиск Декартовым, поэтому его лучше использовать, если у нас есть много элементов, среди которых нужно проводить поиск.

21)Графики сравнения работы операции вставки для отсортированных заранее данных:

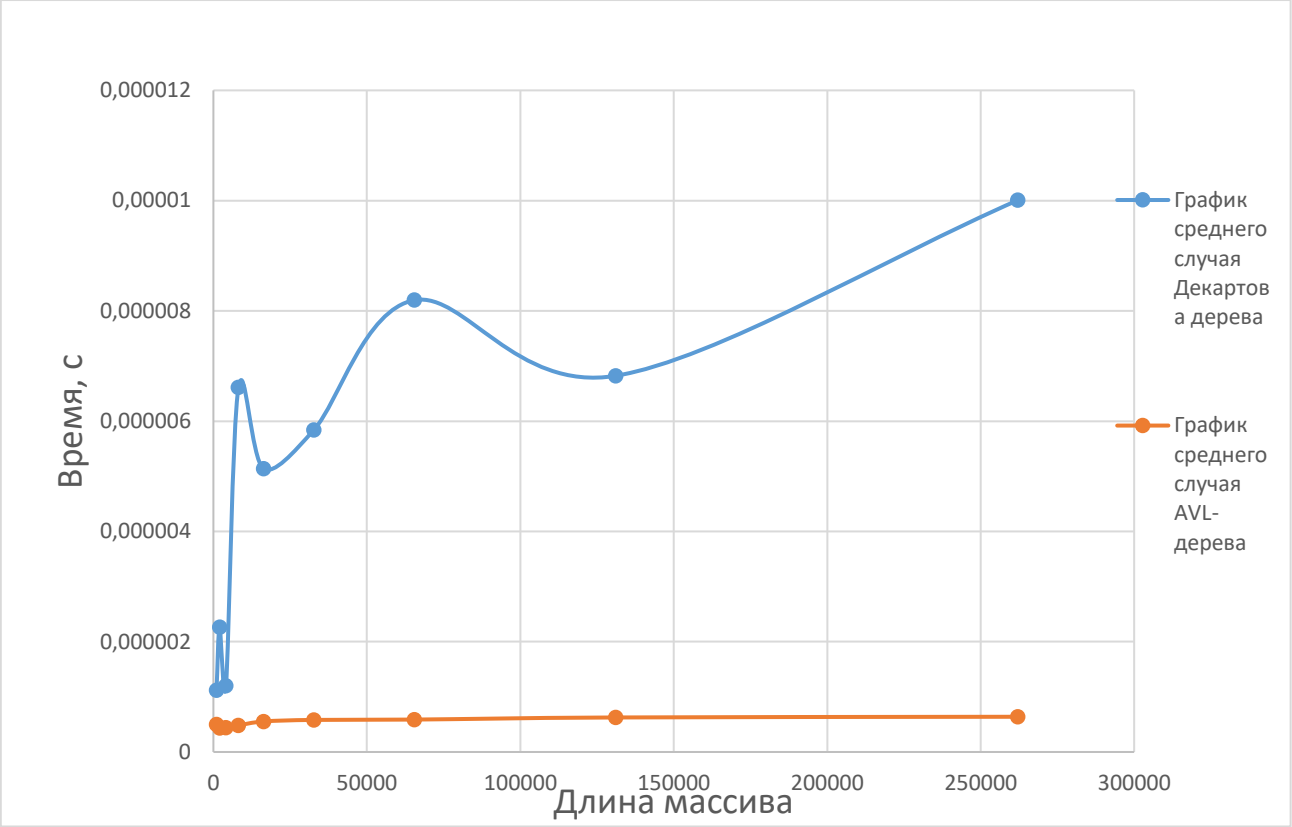

Рис.29. Графики сравнения работы операции вставки AVL и Декартова дерева

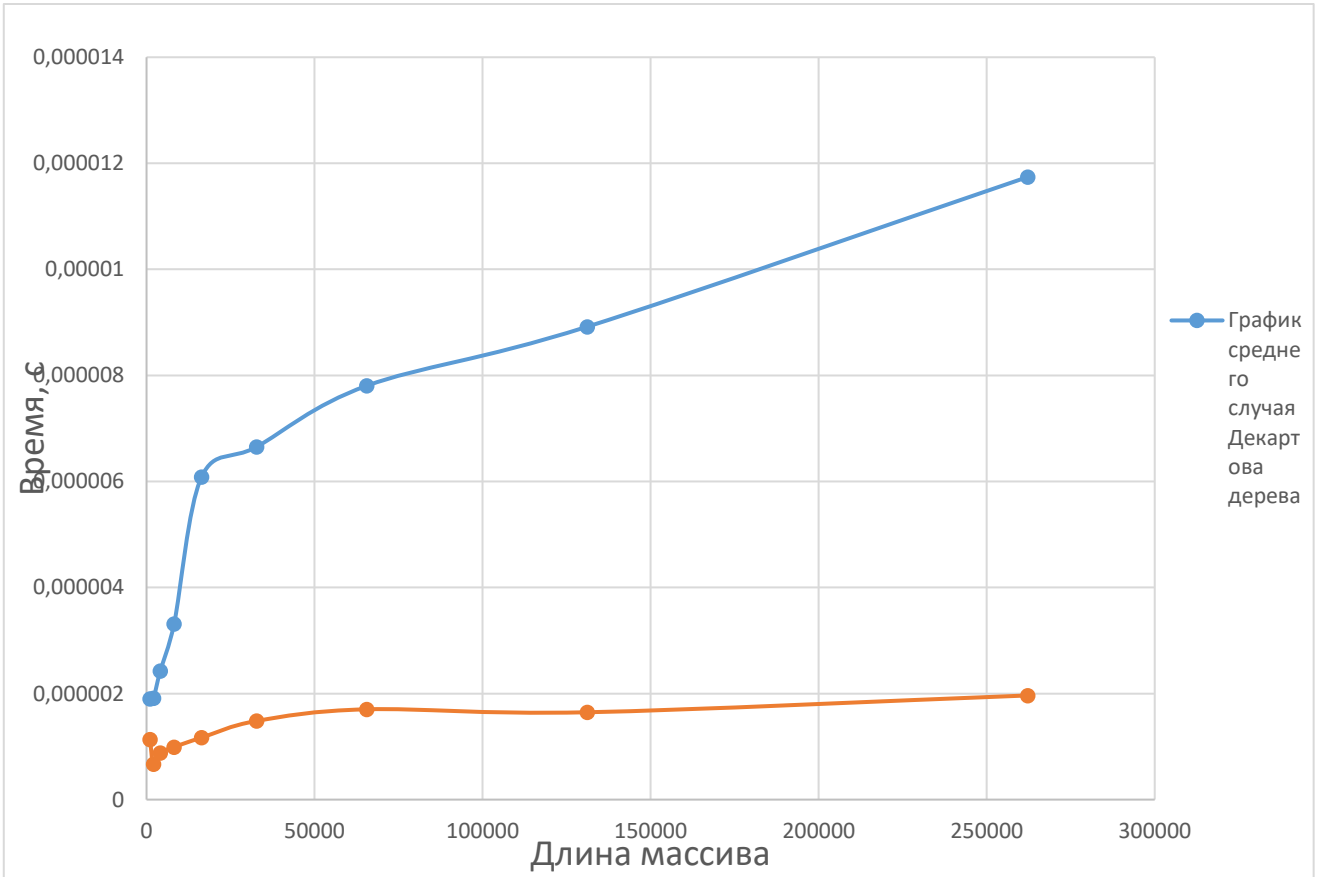

Рис.30. Графики сравнения работы операции удаления AVL и Декартова дерева

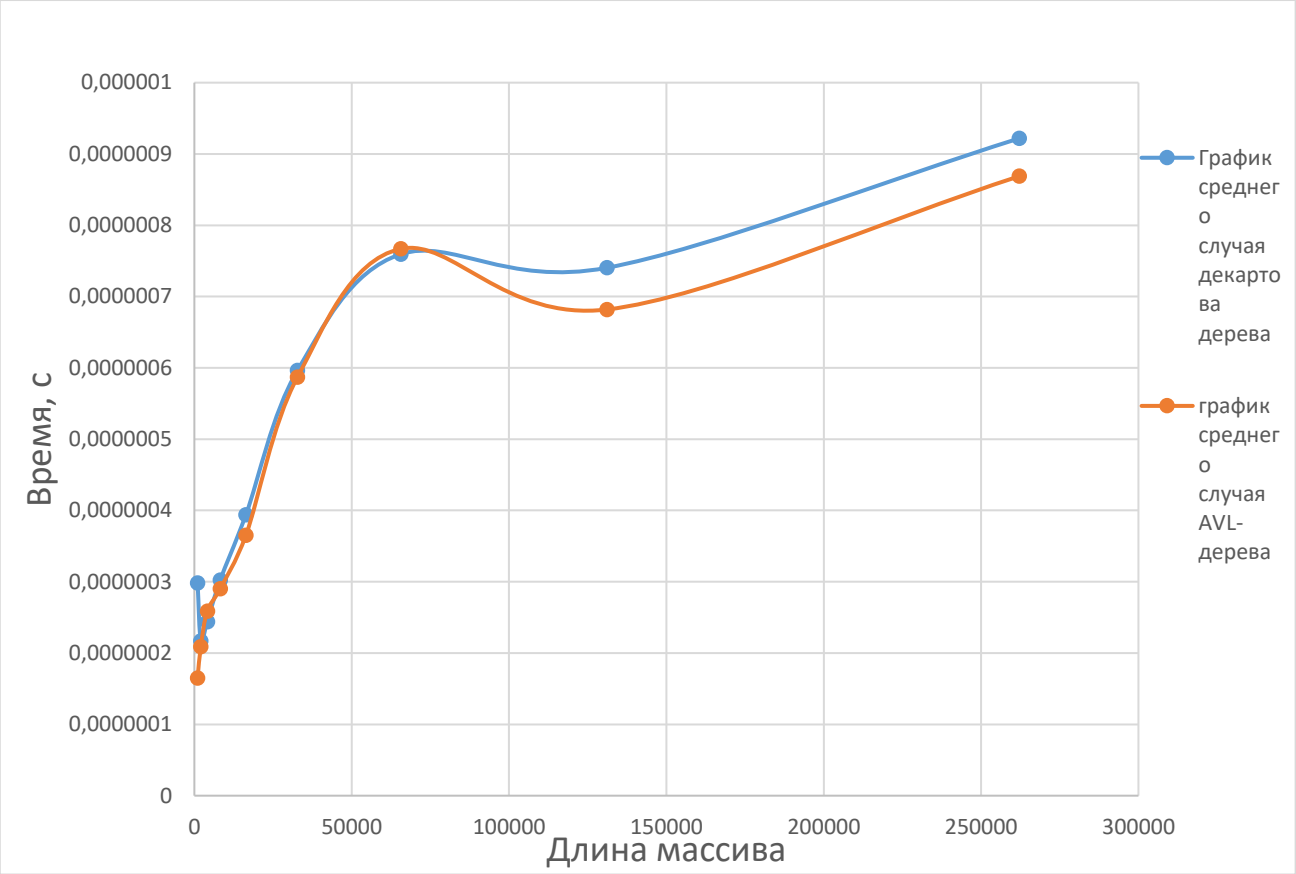

Рис.31. Графики сравнения работы операции удаления AVL и Декартова дерева

На сортированных заранее данных видно, что на Декартово дерево большее влияние оказывают начальные данные.

Заключение.

Правильная Реализация Декартова дерева показалась мне не самой простой, самым сложным для понимания было понять то, как алгоритмы merge и split работают. На графиках можно было увидеть, что время выполнения поиска, вставки и удаления растет вместе с количеством узлов дерева и что с заранее сортированными данными деревья (из-за того, что балансируются) иногда работают незначительно медленнее, чем с неотсортированными. Характер описания всех используемых операций логарифмический и декартово дерево на константное значение в операциях вставки и удаления работает медленнее, зато быстрее при поиске.