ОТЧЕТ ПО ЛАБОРАТОРНОЙ РАБОТЕ № 6
БИНАРНОЕ ДЕРЕВО.AVL

Оглавление

Описание задачи.

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.

Двоичное дерево – это иерархическая структура, в которой каждый узел содержит не более чем двух потомков.

Для каждого узла, тот узел, который стоит выше по иерархии для него называют родительским узлом, а те узлы, что стоят ниже, для которых этот узел является родительским, называются правым и левым наследниками рёбер

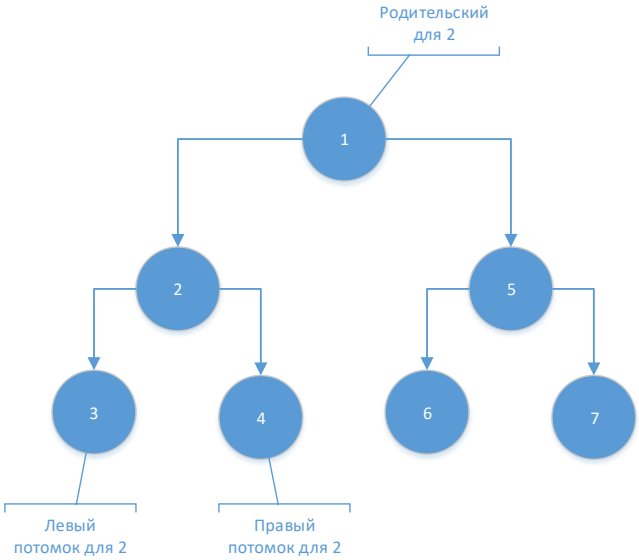

Рис1.Дерево

Существует рекурсивное определение двоичного дерева:

<дерево> = (

<узел>,

<дерево(левое)>, <дерево(правое)>

) | пустота

Т.е. Дерево либо пустое, либо имеет два поддерева, которые в свою очередь тоже либо пустые либо имеют аналогичное устройство.

Вставка O(1)

Удаление O(1)

Двоичное дерево поиска

Список и вектор/массив, но у них есть ряд недостатков связанных в основном с операциями поиска и вставки удаления, так:

- Для списка вставка/удаление имеют сложность O(1), а поиск O(N)
- Для вектора и массива вставка удаление имеют сложность O(N), а поиск O(log(N))

Как видно, у этих структур имеется достоинство только в одном из двух областей, либо в поиске либо в области вставки, как же можно совместить преимущество быстрой вставки/удаления и быстрого поиска.

Именно для этого можно использовать двоичное дерево, превратив его в двоичное дерево поиска, для этого необходимо ввести следующие правила при работе с двоичным деревом:

- Оба поддерева являются двоичными деревьями поиска
- У всех узлов левого поддерева любого узла, хранимое значение всегда меньше либо равно хранимого значения этого узла.
- У всех узлов правого поддерева любого узла, хранимое значение всегда больше либо равно хранимого значения этого узла.

Двоичное дерево поиска – это двоичное дерево придерживающееся 2х правил, согласно которым левые потомки всегда меньше или равны текущего элемента, а правые больше или равны текущего элемента.

Поисковые деревья — это решения так называемой «словарной проблемы». Предположим, что имеется большое количество ключей, каждый из которых имеет значение. В немецко–английском словаре немецкое слово является ключевым, а английские слова являются значением, которое вы ищете. Аналогично ведет себя телефонная книга с именем и адресом в качестве ключа, а номер телефона — в качестве искомого значения.

Этот подход получил в информатике название «двоичный поиск». Она воссоздана очевидным образом с помощью очень известного метода поиска «двоичный поиск в массиве». Их поведение оптимально с точки зрения информации, а именно логарифмически.

Обход дерева

Процесс обхода дерева проще всего реализовать как рекурсивный процесс. Последовательно обследуя каждый узел дерева и посещая сначала левую его часть, затем правую. Стандартный порядок обхода выдаст отсортированный по возрастанию результат, его инверсия отсортированный по убыванию. Так же, можно использовать обход в ширину, тогда получится другой порядок обхода.

Для приведенного дерева вывод будет: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10

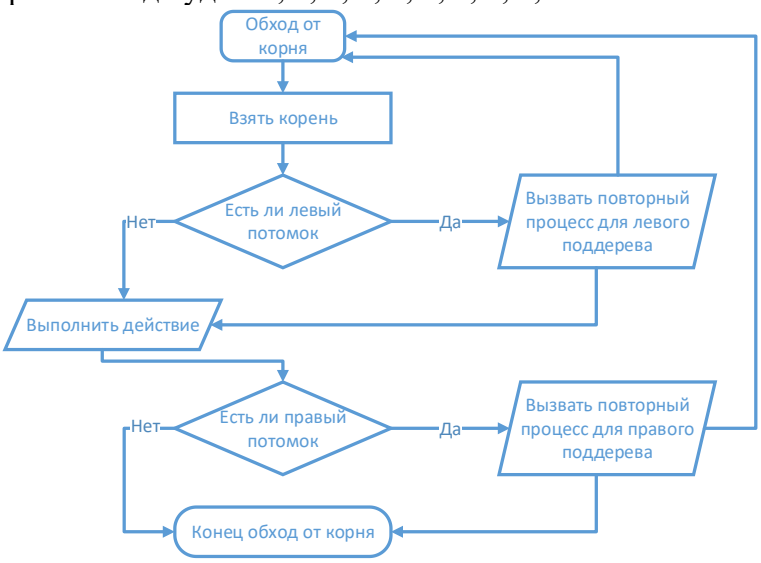

Рис.2.Алгоритм обхода дерева

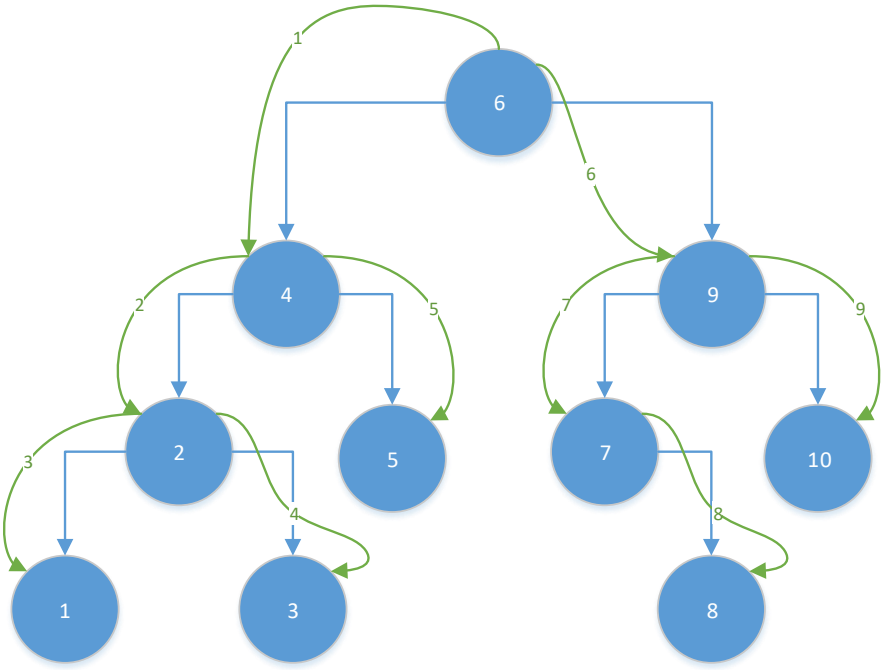

Рис.3.Демонстрация обхода по дереву

Вставка в дерево

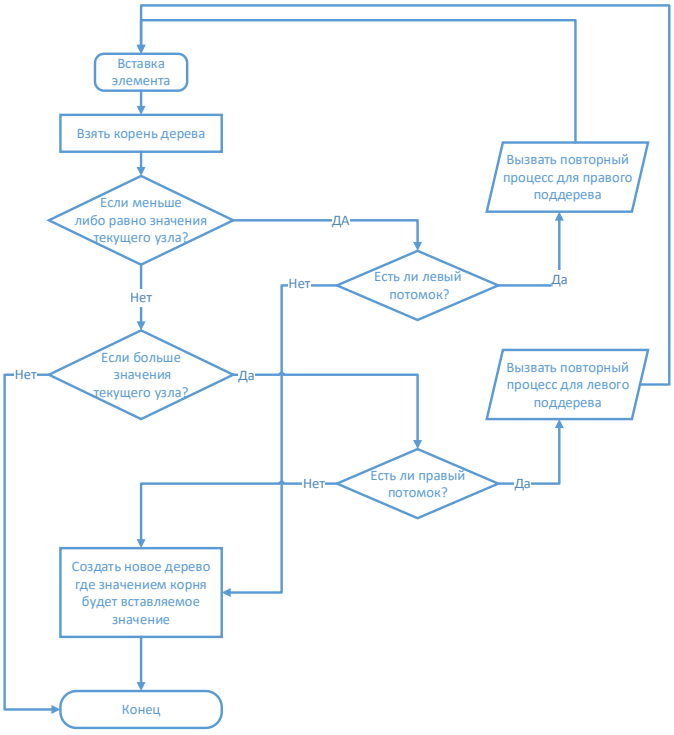

Рис.4.Алгоритм вставки в дерево

Удаление из дерева

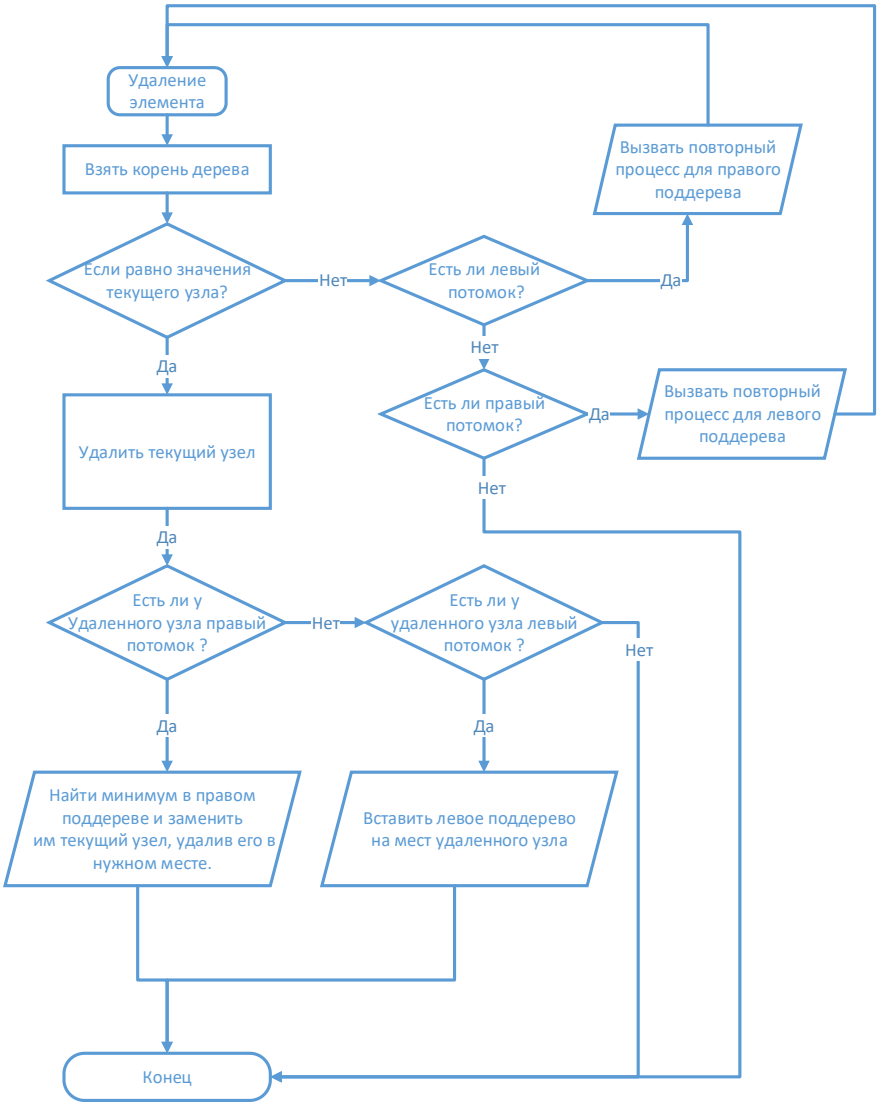

Рис.5 Алгоритм удаления из дерева.

Процесс удаления из дерева так же рекурсивен, особенностью этого процесса является тот факт, что при удалении узла мы должны решить следующие пограничные случаи:

1. Удаляемый узел не имеет потомков. В этом случае узел просто удаляется.
2. Удаляемый узел имеет только левых потомков. В этом случае вместо удаляемого узла становится его левый потомок.
3. Удаляемый узел имеет правых потомков. В этом случае, среди правых потомков требуется найти минимальное значение, т.е. самого левого потомка из правых потомков, удалить его в правом поддереве и вставить на место удаляемого узла, сохранив все связи удаляемого узла.

Самобалансирующиеся деревья

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 построится 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.

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)).

Узел AVL дерева, содержит:

- Ключ
- Высоту – от лепестков
- Левый потомок
- Правый потомок

По поводу высоты, можно учесть важный момент, классически атрибут высоты содержит не высоту, а разницу, которая может быть -1,0,1, так как в остальных случаях вызывается перебалансировка, но из-за логарифмической природы высоты, можно смело хранить само значение, но тогда придется всегда рассчитывать фактор перебалансировки, который будет разницей между высотами правого и левого поддерева.

Балансировка

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.

Малый левый поворот

Малый левый поворот, это операция балансировки, которая является зеркальным отражением малого правого поворота, и предполагает смену верхнего узла на его левого потомка, так, чтобы он сам стал правым потомком замещающего узла.

Малый правый поворот

Взвешенным называют такой граф, каждое ребро которого сопоставимо с каким-либо числовым значением называемым весом графа. Вес ребра в графе может отражать какой-либо параметр в системе, которая этим гарфом описывается. Например, если есть набор пунктов назначения соединённых друг с другом дорогами веса на таком графе могут означать длину этих дорог.

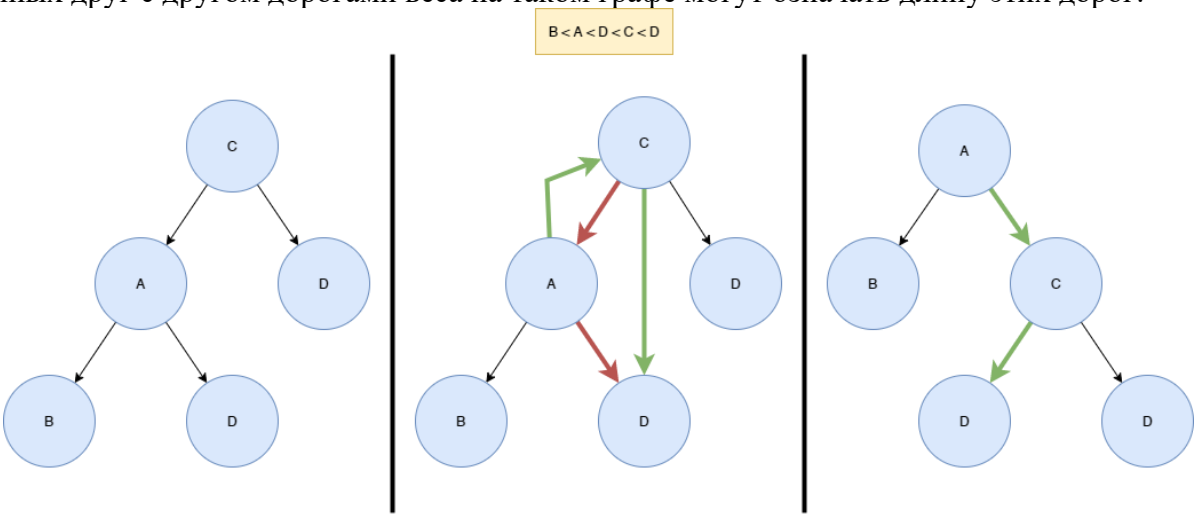

Рис.6. Малый правый поворот

Большой левый поворот

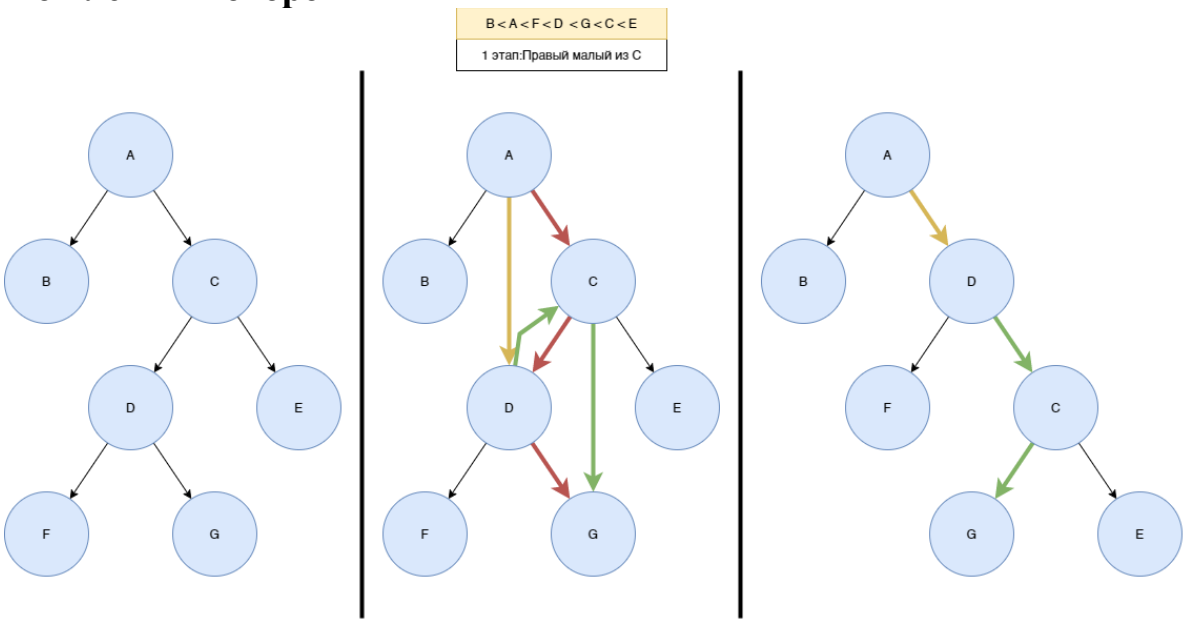

Рис.7. Большой левый поворот 1-этап.

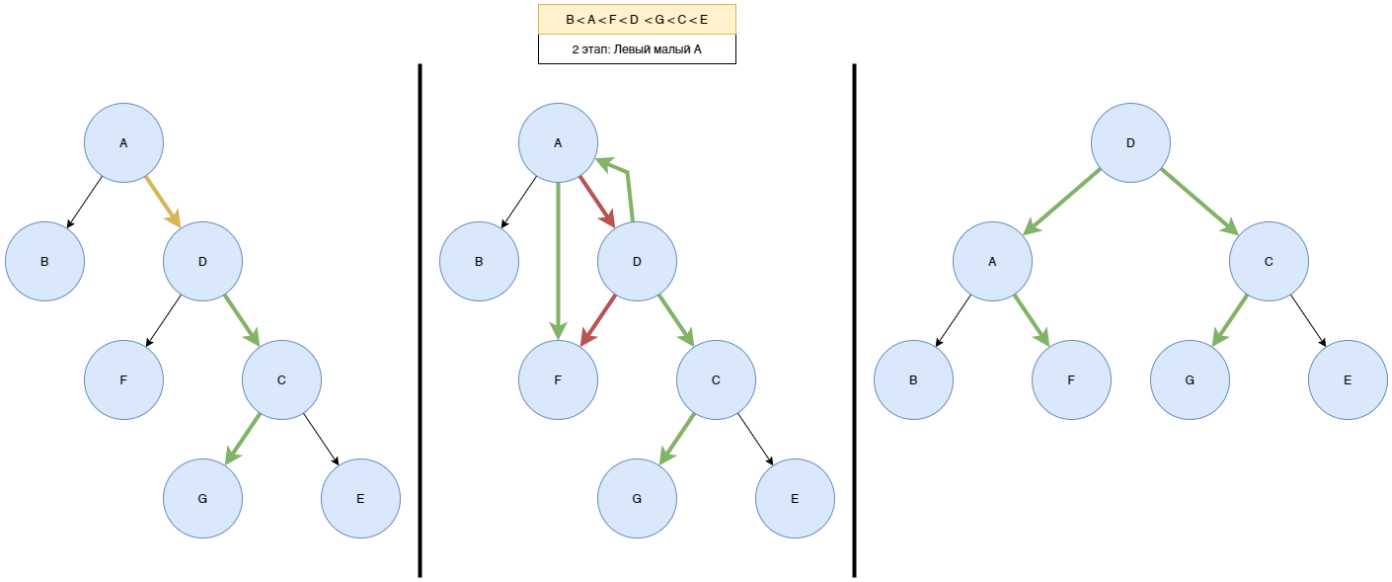

Рис.8. Большой левый поворот 2-этап.

Большой правый поворот

Алгоритм Прима

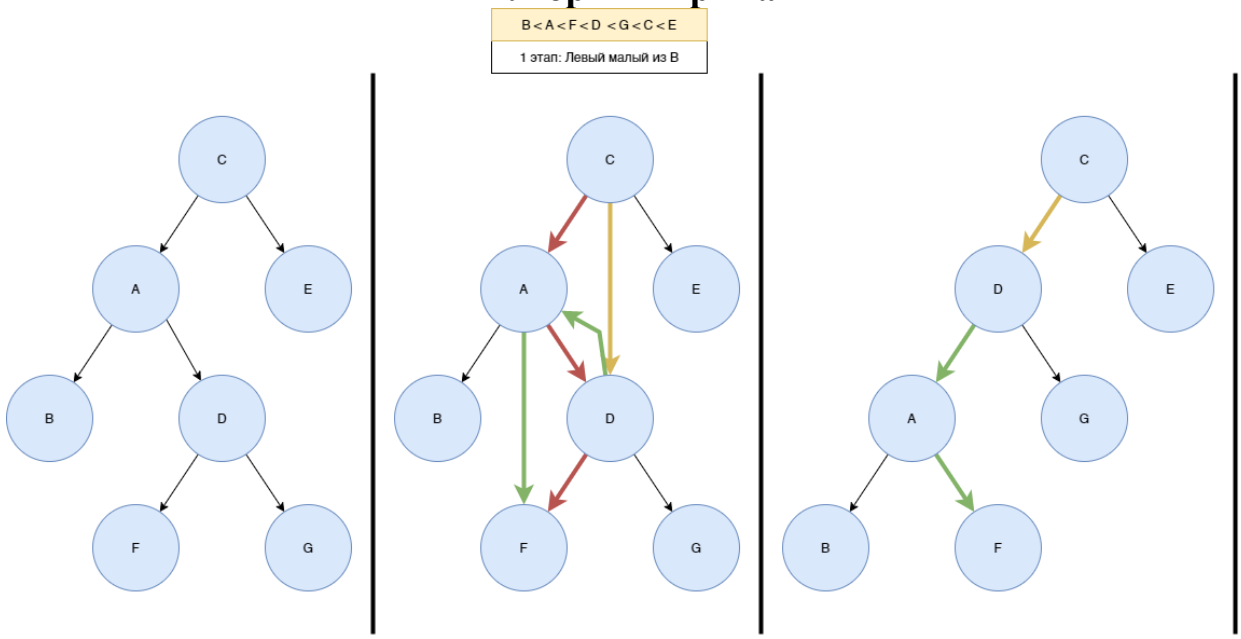

Рис.9. Большой правый поворот 1-этап.

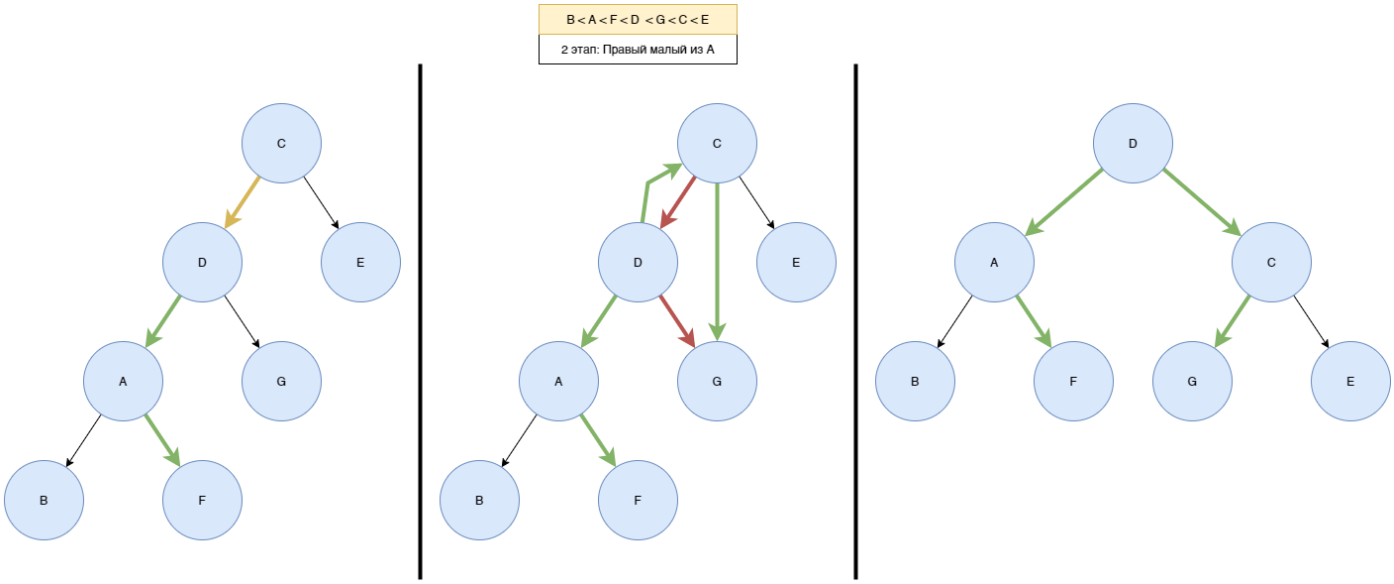

Рис.10. Большой правый поворот 2-этап.

Выполнение задачи.

Программа для получения тестовых данных была написана на C#.

Код:

```
    /// <summary>
/// Бинарное дерево
/// </summary>
class BinTree
{
    public class Node
    {
        public int d_value;
        public Node left;
        public Node right;
        public Node(int value)
        {
            this.d_value = value;
        }
    }
    Node root;//узлы
    //public BinTree()
    //{
    //}
    /// <summary>
    /// Метод добавления элемента в дерево
    /// </summary>
    /// <param name="value">Добавляемый узел</param>
    public void Add(int value)
    {
        Node new_item = new Node(value);
        if (root == null)//если элемент первый
        {
            root = new_item;
        }
        else
        {
            root = insertRoot(root, new_item);
        }
    }
    /// <summary>
    /// Рекурентное добавление нового узла в дерево
    /// </summary>
    /// <param name="current">Текущий узел</param>
    /// <param name="n">Новый узел</param>
    /// <returns>Дерево с новым узлом</returns>
    public virtual Node insertRoot(Node current, Node n)
    {
        if (current == null)//новый узел, если ещё нет
        {
            current = n;
            return current;
        }
        else if (n.d_value < current.d_value)
        {
            current.left = insertRoot(current.left, n);

        }
        else if (n.d_value > current.d_value)
        {
            current.right = insertRoot(current.right, n);

        }
        return current;
    }
    /// <summary>
    /// Удаление корня
```

```
/// </summary>
/// <param name="target"></param>
public void Delete(int target)
{
    root = Delete(root, target);
}
/// <summary>
/// Рекурентное удаление выбранного узла
/// </summary>
/// <param name="current">Текущий дерево</param>
/// <param name="target">Удаляемый корень</param>
/// <returns>Дерево с удаленным корнем</returns>
public virtual Node Delete(Node current, int target)
{
    Node parent;
    if (current == null) return null;
    else
    {
        //при поиске узла меньше
        if (target < current.d_value)
        {
            current.left = Delete(current.left, target);

        }
        //при поиске узел больше
        else if (target > current.d_value)
        {
            current.right = Delete(current.right, target);

        }
        //когда узел найден найден
        else
        {
            if (current.right != null)
            {
                //если есть правый потомок, заменяем самым левым
                parent = current.right;
                while (parent.left != null)
                {
                    parent = parent.left;
                }
                current.d_value = parent.d_value;
                current.right = Delete(current.right, parent.d_value);

            }
            //если есть левый потомок, заменяем им
            else if (current.left != null)
            {   //if current.left != null
                return current.left;
            }
            //если потомков нет, так и оставляем
            else
            {
                return null;
            }
        }
    }
    return current;
}
/// <summary>
/// Метод нахождения узла
/// </summary>
/// <param name="el">узел</param>
public void Find(int el)
{
    Node find = Find(root, el);
    if (find != null && find.d_value == 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.d_value)
        {
            if (target == current.d_value)
            {
                return current;
            }
            else
                return Find(current.left, target);
        }
        else
        {
            if (target == current.d_value)
            {
                return current;
            }
            else
                return Find(current.right, target);
        }

    }
    /// <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)
        {
            depth++;
            printTreeOrder(current.left, depth);
            Console.Write($"{depth}-({current.d_value})\n");
            printTreeOrder(current.right, depth);
        }
    }
}
/// <summary>
/// AVL-дерево
/// </summary>
class AVL : BinTree
{
```

```
/// <summary>
/// Рекурентное добавление нового узла в дерево с балансировкой
/// </summary>
/// <param name="current">Текущий узел</param>
/// <param name="n">Новый узел</param>
/// <returns>Дерево с новым узлом</returns>
public override Node insertRoot(Node current, Node n)
{
    if (current == null)
    {
        current = n;
        return current;
    }
    else if (n.d_value < current.d_value)
    {
        current.left = insertRoot(current.left, n);
        current = balance(current);//балансировка
    }
    else if (n.d_value > current.d_value)
    {
        current.right = insertRoot(current.right, n);
        current = balance(current);
    }
    return current;
}
/// <summary>
/// Балансировка Бинарного дерева
/// </summary>
/// <param name="current">Текущий узел</param>
/// <returns>Сбалансированный узел</returns>
public Node balance(Node current)
{
    int b_factor = BalanceFactor(current);
    if (b_factor > 1)//если по левой
    {
        if (BalanceFactor(current.left) > 0)//если по левой и левое поддерево больше
        {
            current = rotateSR(current);
        }
        else//если по левой и правое поддерево больше
        {
            current = rotateBL(current);
        }
    }
    else if (b_factor < -1)//если по правой
    {
        if (BalanceFactor(current.right) > 0)
        {
            current = rotateBR(current);//если по правой и левое поддерево больше
        }
        else
        {
            current = rotateSL(current);////если по правой и правое поддерево больше
        }
    }
    return current;
}
/// <summary>
/// Рекурентное удаление выбранного узла с балансировкой
/// </summary>
/// <param name="current">Текущий дерево</param>
/// <param name="target">Удаляемый корень</param>
/// <returns>Дерево с удаленным корнем</returns>
public override Node Delete(Node current, int target)
{
    Node parent;
    if (current == null) return null;
    else
    {
```

```
        //при поиске узла меньше
        if (target < current.d_value)
        {
            current.left = Delete(current.left, target);
            current = balance(current);
        }
        //при поиске узла больше
        else if (target > current.d_value)
        {
            current.right = Delete(current.right, target);
            current = balance(current);
        }
        //когда узел найден
        else
        {
            if (current.right != null)
            {
                //если есть правый потомок, заменяем самым левым
                parent = current.right;
                while (parent.left != null)
                {
                    parent = parent.left;
                }
                current.d_value = parent.d_value;
                current.right = Delete(current.right, parent.d_value);
                current = balance(current);
            }
            else if (current.left != null)//если есть левый потомок, заменяем им
            {   //if current.left != null
                return current.left;
            }
            else//если потомков нет, так и оставляем
            {
                return null;
            }
        }
    }
    return current;
}
/// <summary>
/// Получение высоты
/// </summary>
/// <param name="current">Текущий узел</param>
/// <returns>Величина высоты</returns>
private int getHeight(Node current)
{
    int height = 0;
    if (current != null)
    {
        int l = getHeight(current.left);//по левой стороне
        int r = getHeight(current.right);//по правой стороне
        int m = Math.Max(l, r);
        height = m + 1;
    }
    return height;
}
/// <summary>
/// Фактор балансировки
/// </summary>
/// <param name="current">Текущий узел</param>
/// <returns>Значения фактора балансировки</returns>
private int BalanceFactor(Node current)
{
    int l = getHeight(current.left);
    int r = getHeight(current.right);
    int b_factor = l - r;
    return b_factor;
}
/// <summary>
```

```
    /// Меод малого левого поворота
    /// </summary>
    /// <param name="parent">Узел-Родитель</param>
    /// <returns>Дерево</returns>
    private Node rotateSL(Node parent)
    {
        Node pivot = parent.right;
        parent.right = pivot.left;
        pivot.left = parent;
        return pivot;
    }
    /// <summary>
    /// Малый правый поворот
    /// </summary>
    /// <param name="parent">Узел-родитель</param>
    /// <returns>Дерево</returns>
    private Node rotateSR(Node parent)
    {
        Node pivot = parent.left;
        parent.left = pivot.right;
        pivot.right = parent;
        return pivot;
    }
    /// <summary>
    /// Большой левый поворот
    /// </summary>
    /// <param name="parent">Узел-родитель</param>
    /// <returns>Дерево</returns>
    private Node rotateBL(Node parent)
    {
        Node pivot = parent.left;
        parent.left = rotateSL(pivot);
        return rotateSR(parent);
    }
    /// <summary>
    /// Большой правый поворот
    /// </summary>
    /// <param name="parent">Узел-родитель</param>
    /// <returns>Дерево</returns>
    private Node rotateBR(Node parent)
    {
        Node pivot = parent.right;
        parent.right = rotateSR(pivot);
        return rotateSL(parent);
    }
}
```

Реализация:

1)Бинарное дерево представляет собой класс BinTree с полями root(всё дерево) и подклассом Node с полями value(хранит значение узла), left (хранит левого потомка), right(правый потомок).

2) В методе Add () я добавляю новые узлы. Проверяю для начала есть ли в дереве элементы, чтобы было к чему добавить, а после рекурсивно добавляю в методе insertRoot (). Этот метод работает по описаному выше алгоритму: смотрим наше значение меньше или больше текущего узла, если меньше или больше, то соответственно опускаемся в его потомков слева и справа и так до момента, когда не найдется потомков меньше/больше нашего.

3) В методе Delete () я рекурсивно обхожу дерево и удаляю нужный узел по алгоритму:

1. Удаляемый узел не имеет потомков. В этом случае узел просто удаляется.
2. Удаляемый узел имеет только левых потомков. В этом случае вместо удаляемого узла становится его левый потомок.
3. Удаляемый узел имеет правых потомков. В этом случае, среди правых потомков требуется найти минимальное значение, т.е. самого левого потомка из правых потомков, удалить его в правом поддереве и вставить на место удаляемого узла, сохранив все связи удаляемого узла.

5)В методе Find () я рекурсивно обхожу дерево в поисках заданного узла. Последовательно обследуя каждый узел дерева и посещая сначала левую его часть, затем правую.

6)Метод printTree запускает вывод дерево в консоль, а метод printTreeOrder рекурсивно обходит массив слева-направо и выводит узлы в консоль.

7)AVL дерево наследуется от бинарного дерева. Его особенность в том, что оно балансирует бинарное дерево. В методах Add, Delete () я дополнительно запускаю метод balance. В ней, через метод balanceFactor (), в котором рекурсивно подсчитывается разница между левыми поддеревьями и правыми поддеревьями, я узнаю фактор баланса. Если фактор баланса положительный, значит перевес у левых поддеревьев и значит, если еще и у нашего узла наибольшее дерево слева, мы делаем большой правый поворот, перекидывая вправо, иначе (если у текущего узла поддерево справа, совершаем большой левый поворот). Если фактор баланса отрицательный, значит перевеус у правых поддеревьев и значит, если еще у нашего узла наибольшее дерево справа, по аналогии с левым, совершаем малый левый поворот и иначе, большой правый.

8)rotateSR(малый правый)-перекидывает с левого угла на правый, rotateSL(малый левый)-перекидывает поддерево с правого угла на левый. RotateBL (большой левый поворот)-сначала поворачиваем направо, а потом налево, rotateBR(большой малый поворот)-наоборот.

9)В функции Testing() происходит само тестирование:

В каждой серии тестов выполняется 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.

- Создается массив состоящий из 2^(10 + i) элементов, где i это номер серии.
- Массив должен быть помещен в оба вариант двоичных деревьев. При этом замеряется время затраченное на всю операцию вставки всего массива.
- После заполнения массива, выполняется 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.
- Проводится 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.

Результаты:

1)Демонстрация работы дерева.

Вывод Бинарного дерева с добавленными элементами в порядке 6, 9, 7, 10, 8, 4, 2, 1, 3, 5, 4:

```
      6
    4--9
  2--5 7--10
1--3     --8
```

Вывод того же дерева с удаленной четверкой:

```
      6
    5--9
  2--   7--10
1--3     --8
```

Вывод AVL-дерева с добавленными элементами в порядке 6, 9, 7, 10, 8, 4, 2, 1, 3, 5, 4:

```
      7
    4--9
  2--6 8--10
1--3 5--
```

Вывод того же дерева с удаленной четверкой:

```
      7
    5--9
  2--6 8--10
1--3
```

2) Замерил время генерацию несортированного Бинарного Дерева

Таблица 1

Время генерации несортированного бинарного дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.000147	0.000418	0.000205
11	0.000294	0.00424	0.000873
12	0.000674	0.000772	0.000707
13	0.001517	0.001784	0.001644
14	0.003536	0.007098	0.004382
15	0.00835	0.023447	0.011217
16	0.018836	0.048221	0.026762
17	0.050423	0.107606	0.079399
18	0.124634	0.169738	0.14427

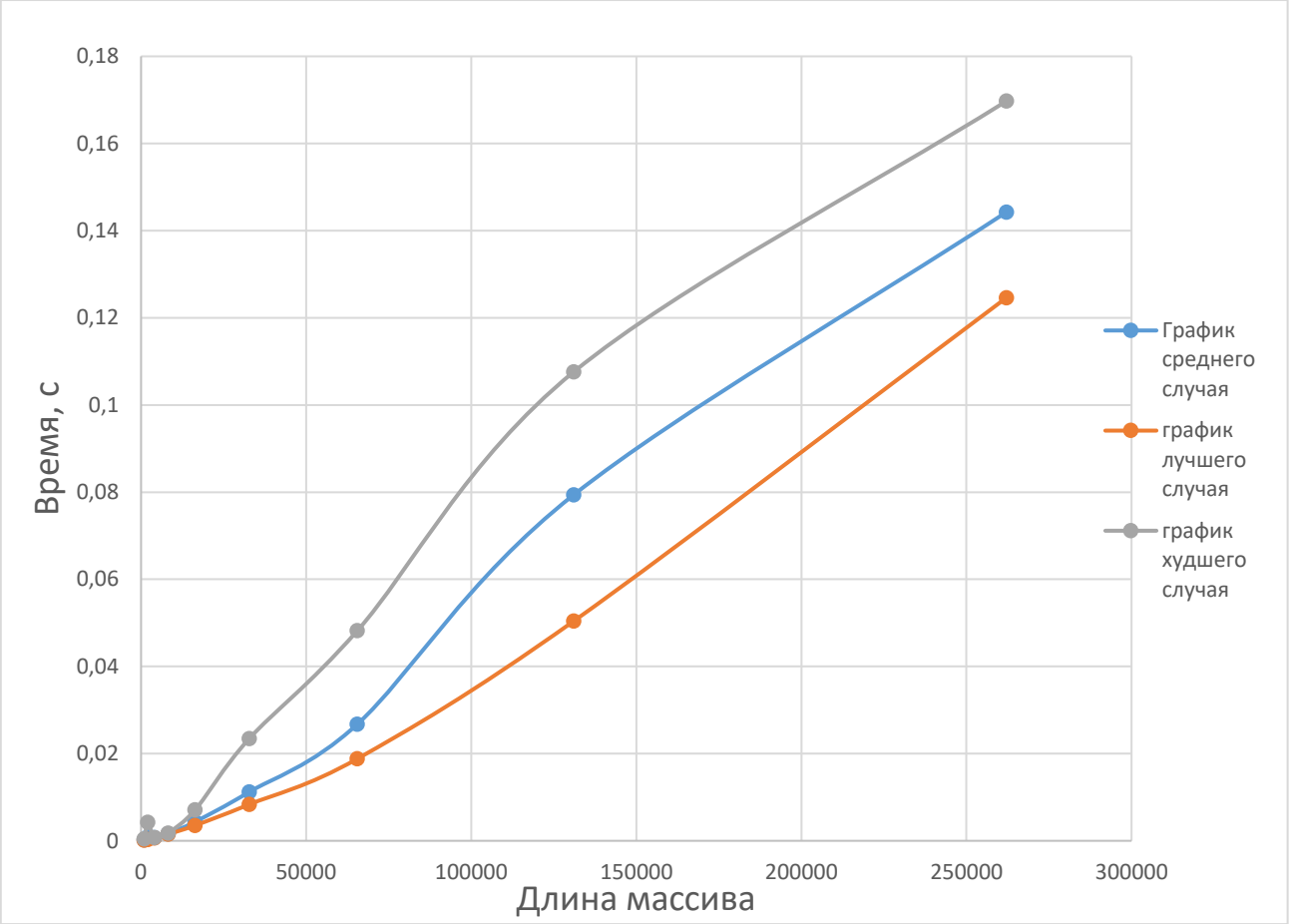

Рис.11. График времени, за которое заполняется несортированное Бинарное дерево

3) Замерил время генерацию сортированного Бинарного Дерева

Таблица 2

Время генерации сортированного бинарного дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.00443	0.008617	0.006041
11	0.018621	0.033333	0.02033
12	0.080392	0.084665	0.082424
13	0.34943	0.355597	0.351359
14	1.459558	1.577049	1.484044
15	6.267311	7.831645	6.7056
16	34.67258	45.00522	40.52932
17	201.4685	249.8989	215.0011
18	918.9456	1072.685	990.4294

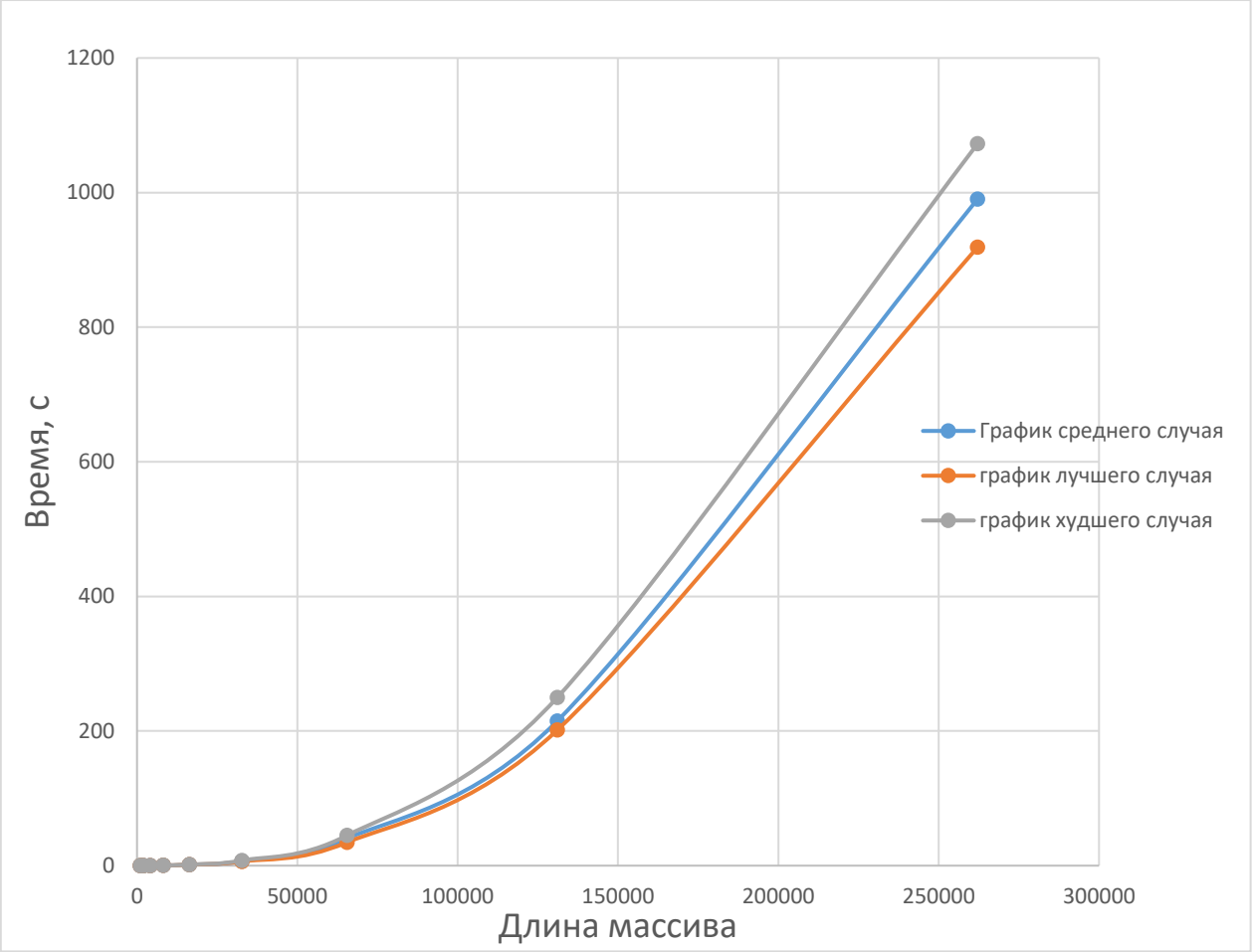

Рис.12. График времени, за которое заполняется сортированное Бинарное дерево

По графикам видно, что отсортированное бинарное дерево с большим кол-вом элементов работает медленнее, чем несортированный.

4) Замерил время генерацию несортированного AVL-дерева

Таблица 3

Время генерации несортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.009928	0.021958	0.013754
11	0.040376	0.06829	0.050531
12	0.190878	0.211606	0.202518
13	0.815187	0.873284	0.84806
14	3.825404	4.0937	3.946832
15	17.21057	18.27988	17.71891
16	77.57316	91.87612	83.06724
17	308.4653	370.6738	339.8291
18	1661.081	5045.74	2129.75

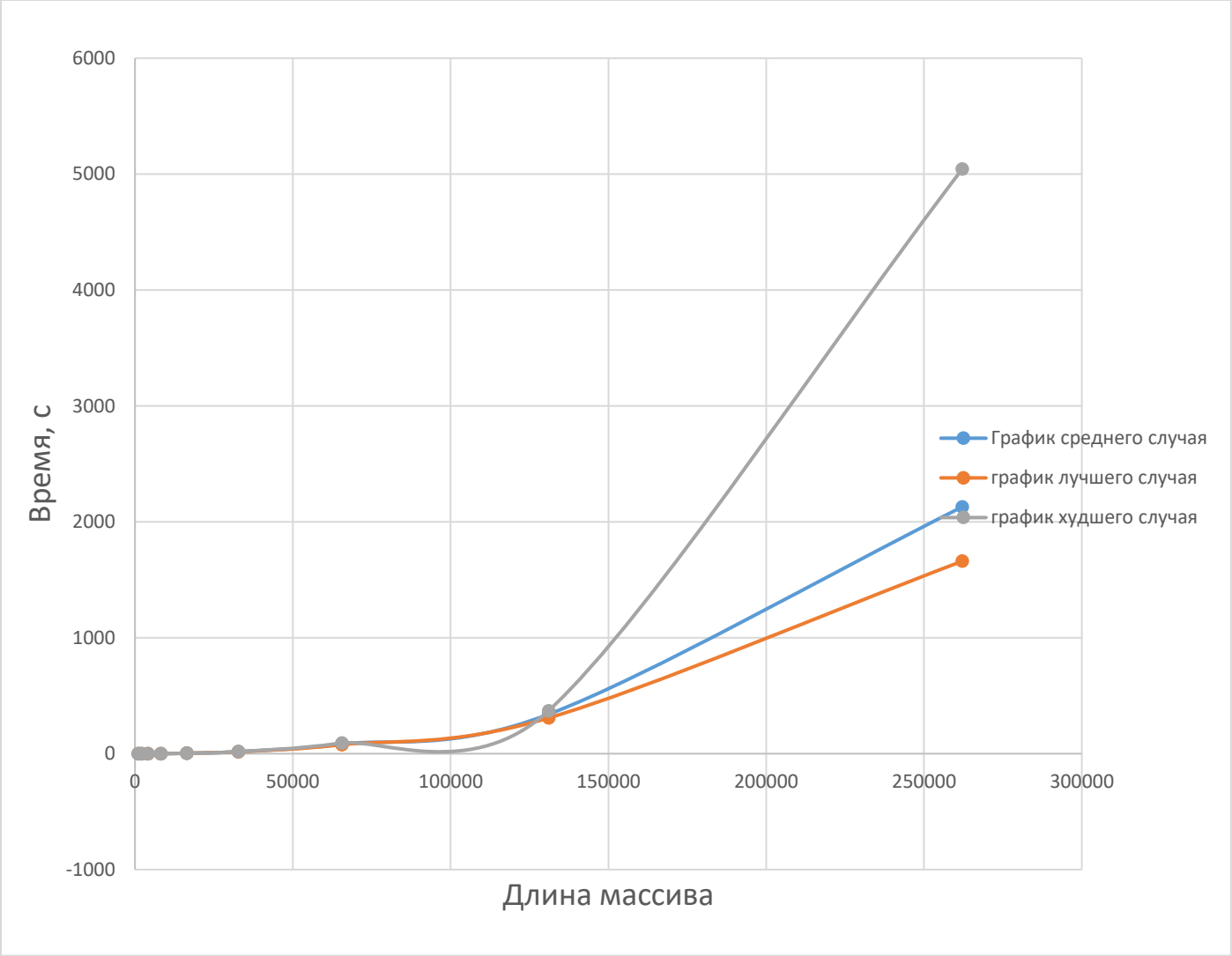

Рис.13. График времени, за которое заполняется несортированное AVL-дерево

5) Замерил время генерацию сортированного AVL-дерева

Таблица 4

Время генерации сортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	0.012048	0.023376	0.016865
11	0.0471	0.066485	0.051611
12	0.187223	0.204022	0.195224
13	0.752118	0.784706	0.76739
14	3.094009	3.467445	3.174646
15	12.70148	14.02504	13.13249
16	53.52745	58.53765	55.53016
17	208.1835	241.9278	227.6519
18	950.2844	3659.507	1537.403

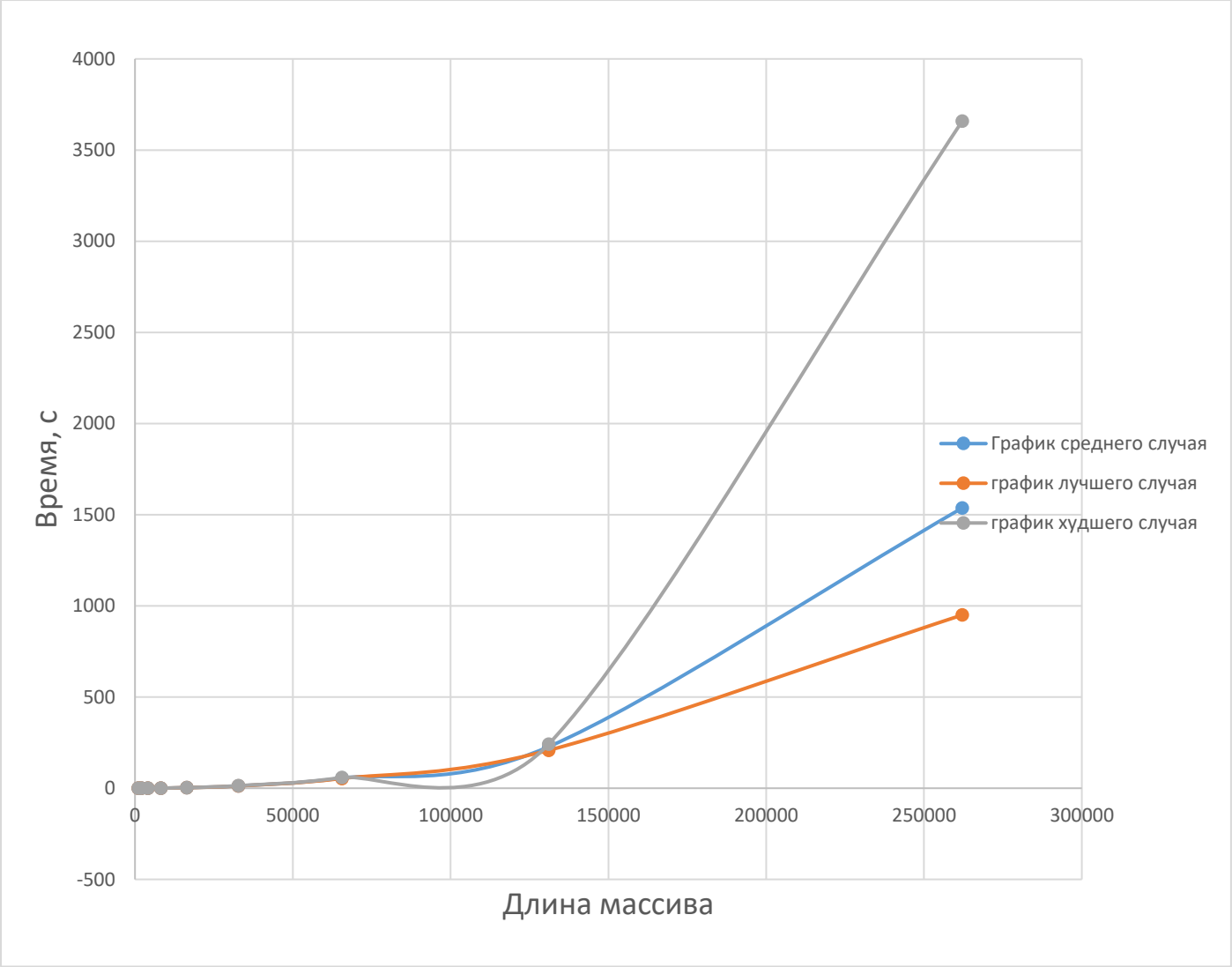

Рис.14. График времени, за которое заполняется сортированное AVL-дерево

По графикам видно, что отсортированное AVL-дерево с большим кол-вом элементов работает медленнее, чем несортированный, но заполнение даже несортированного AVL-дерева выполняется медленнее заполнение несортированного бинарного дерева.

6) Замерил поиск у несортированного Бинарного дерева

Таблица 5

Время поиска одного элемента у несортированного Бинарного дерева в секундах

2^	Лучшее	Худшее	Среднее
10	1.48E-07	4.12E-07	1.88E-07
11	1.72E-07	1.94E-07	1.79E-07
12	2.03E-07	2.34E-07	2.11E-07
13	2.38E-07	2.67E-07	2.56E-07
14	3.16E-07	3.61E-07	3.38E-07
15	3.89E-07	1.02E-06	4.77E-07
16	5.07E-07	6.47E-07	5.48E-07
17	6.20E-07	8.19E-07	7.42E-07
18	1E-06	7.82E-06	2.78E-06

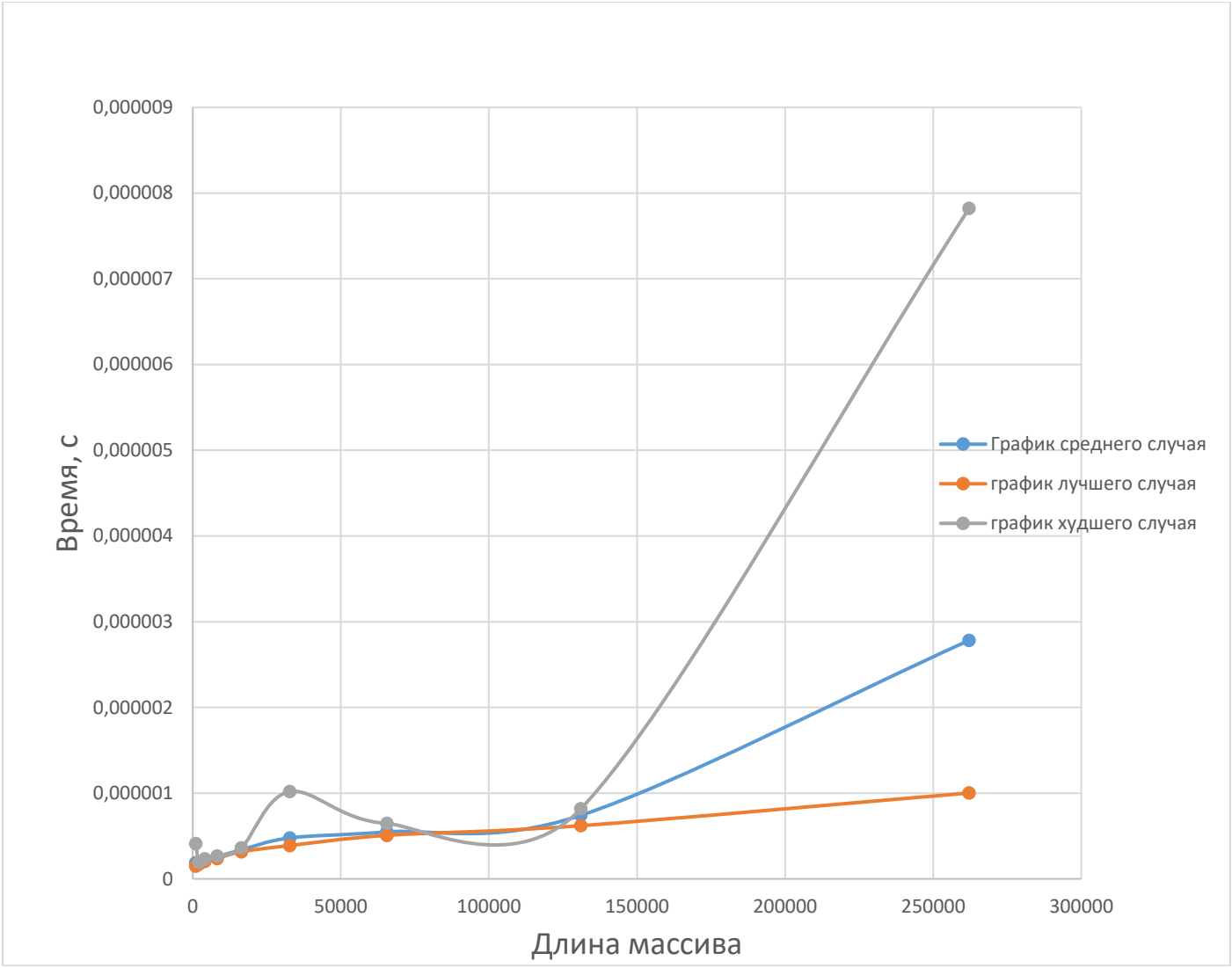

Рис.15. График времени, за которое ищется узел в несортированном Бинарном дереве

7) Замерил поиск у сортированного Бинарного дерева

Таблица 6

Время поиска одного элемента у сортированного Бинарного дерева в секундах

2^	Лучшее	Худшее	Среднее
10	3.79E-06	1.18E-05	5.19E-06
11	7.84E-06	1.29E-05	8.67E-06
12	1.62E-05	1.78E-05	1.71E-05
13	3.60E-05	3.84E-05	3.68E-05
14	7.41E-05	8.58E-05	7.77E-05
15	1.59E-04	1.79E-04	1.67E-04
16	4.09E-04	6.18E-04	4.68E-04
17	1.12E-03	0.001456	0.00127
18	0.002774	0.003648	0.003123

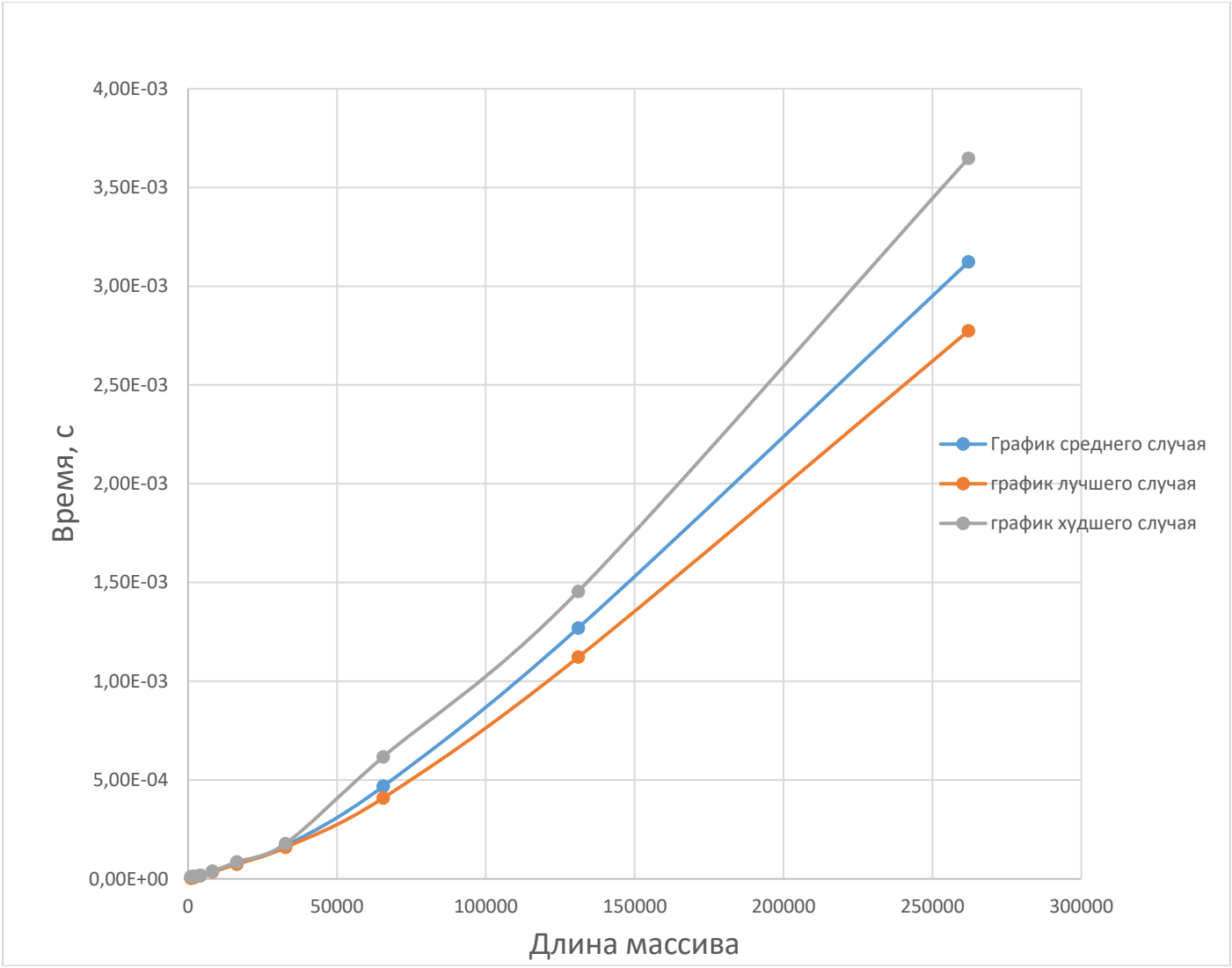

Рис.16. График времени, за которое ищется узел в сортированном Бинарном дереве

По графикам видно, что поиск бинарным деревом с большим кол-вом элементов работает медленнее, чем несортированным.

8) Замерил поиск у несортированного AVL-дерева

Таблица 7

Время поиска одного элемента у несортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	1.33E-07	2.41E-07	1.57E-07
11	1.5E-07	2.63E-07	1.67E-07
12	1.71E-07	1.86E-07	1.81E-07
13	2.23E-07	2.32E-07	2.29E-07
14	2.89E-07	3.18E-07	3.01E-07
15	3.27E-07	3.93E-07	3.75E-07
16	4.48E-07	5.46E-07	4.83E-07
17	5.60E-07	7.36E-07	6.41E-07
18	7.62E-07	1.22E-06	9.67E-07

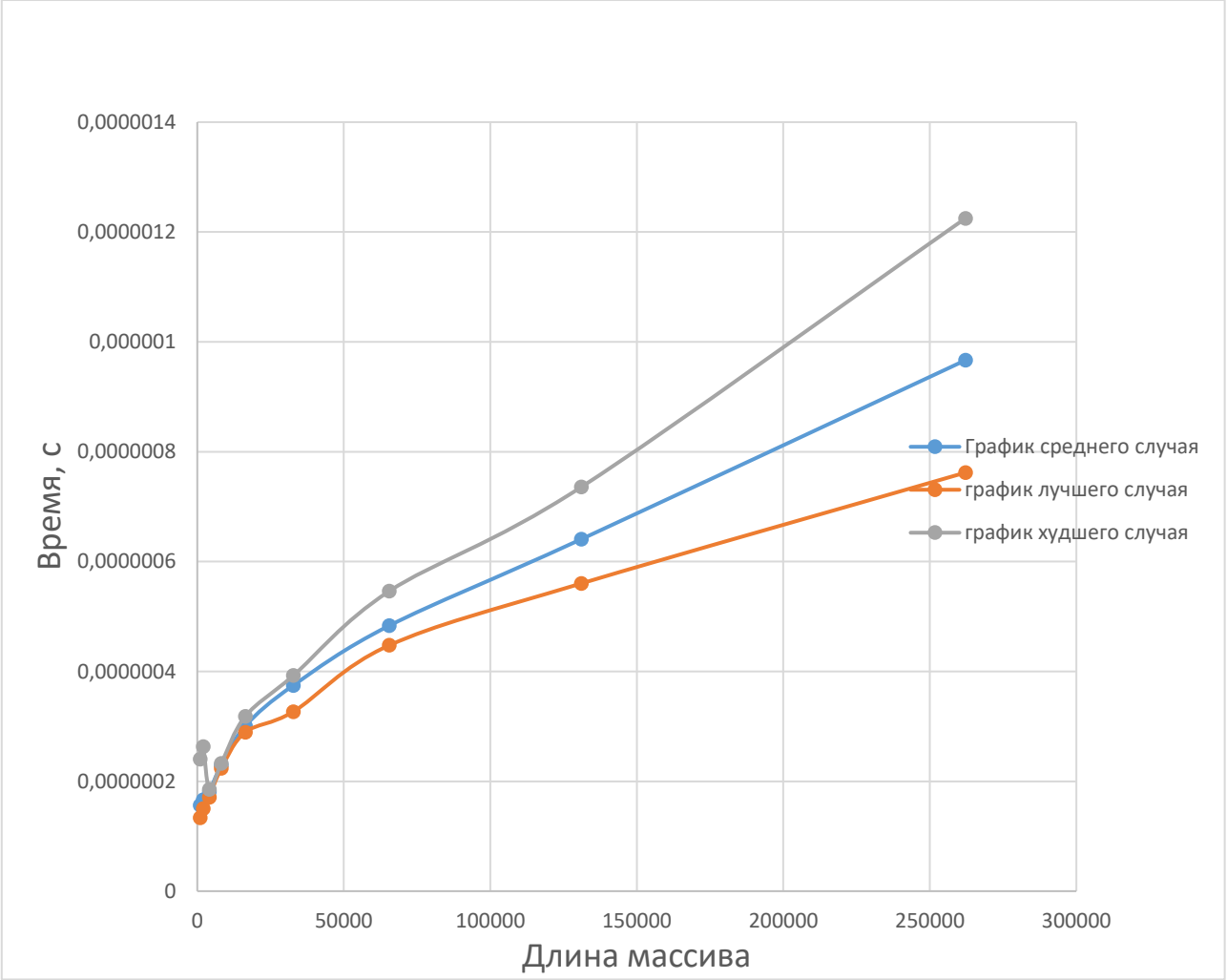

Рис.17. График времени, за которое ищется узел в несортированном AVL-дереве

9) Замерил поиск у сортированного AVL-дерева

Таблица 8

Время поиска одного элемента у сортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	1.26E-07	3.11E-07	1.74E-07
11	1.39E-07	2.18E-07	1.54E-07
12	1.61E-07	1.79E-07	1.72E-07
13	1.87E-07	3.01E-07	2.14E-07
14	2.71E-07	4.55E-07	2.95E-07
15	3.48E-07	3.91E-07	3.59E-07
16	4.42E-07	4.94E-07	4.61E-07
17	5.42E-07	7.2E-07	5.82E-07
18	6.42E-07	1.96E-06	7.97E-07

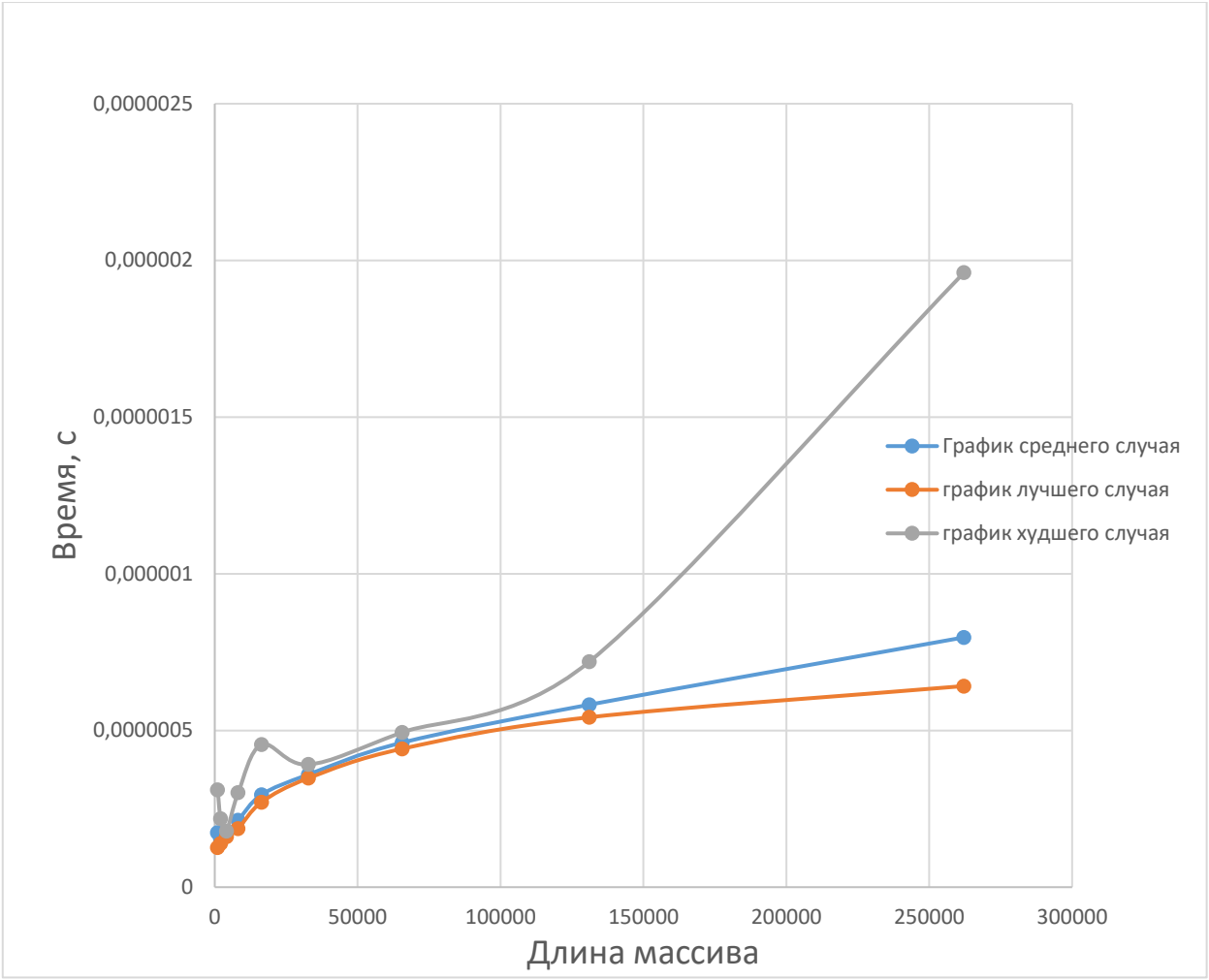

Рис.18. График времени, за которое ищется узел в сортированном AVL-дереве

По графикам видно, что поиск у AVL-дерева с большим кол-вом элементов работает немного медленнее, чем у несортированного, но оба гораздо быстрее поиска по бинарному дереву.

10) Замерил поиск в сравнении с несортированным массивом

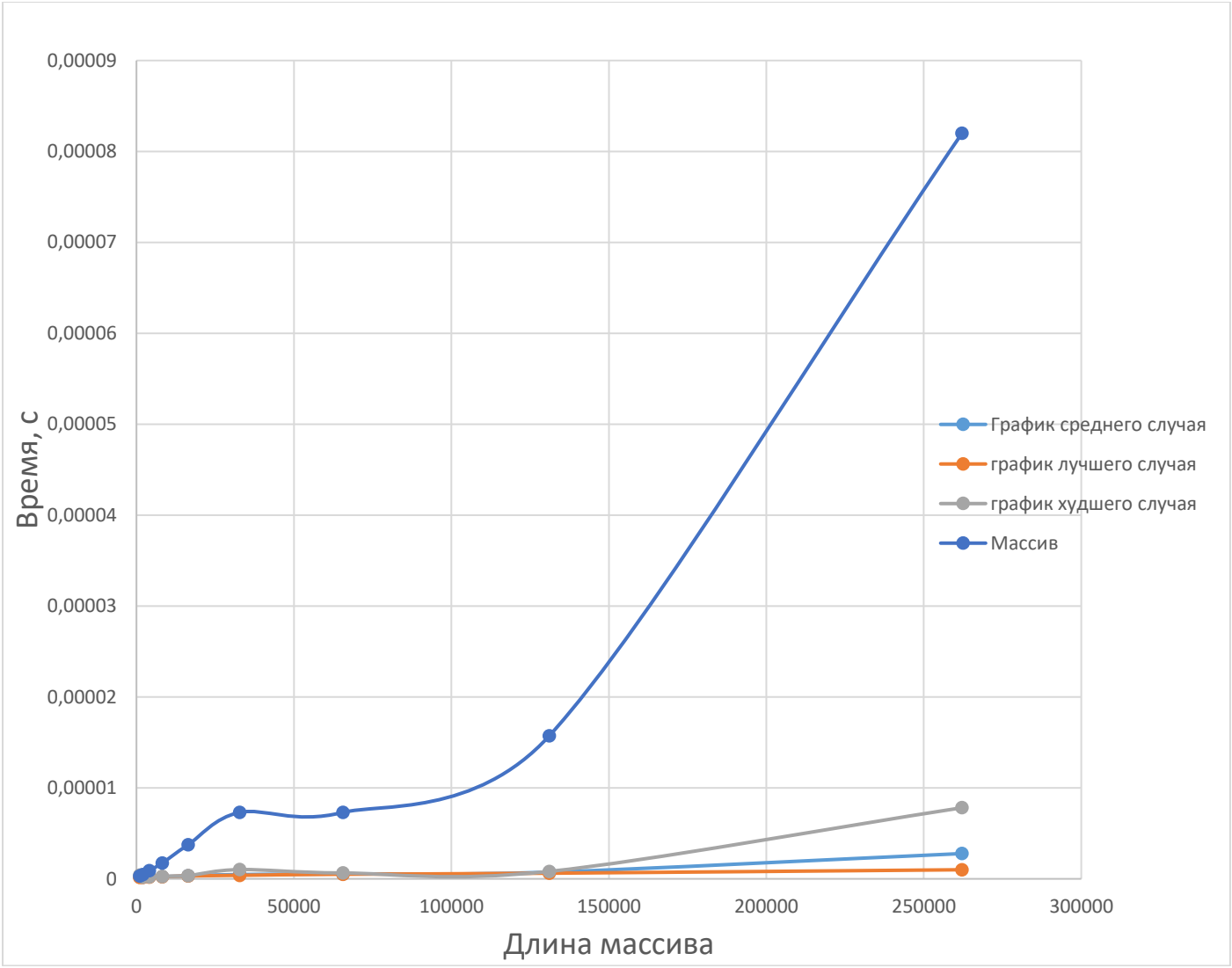

Рис.19.1 График времени, за которое ищется узел в Бинарном дереве в сравнении с несортированным массивом

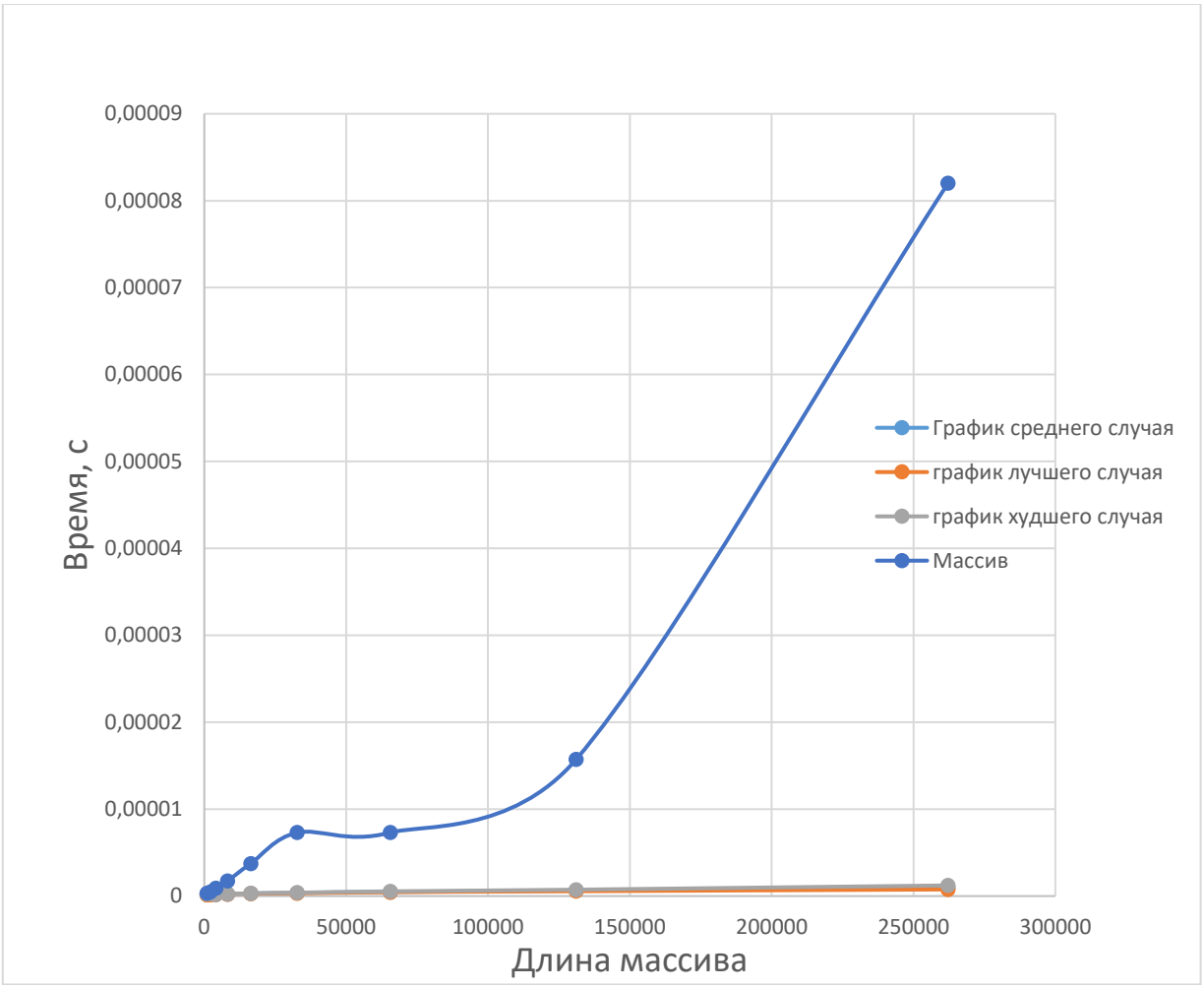

Рис.19.2. График времени, за которое ищется узел в AVL-дереве в сравнении с несортированным массивом

11) Замерил поиск в сравнении с сортированным массивом

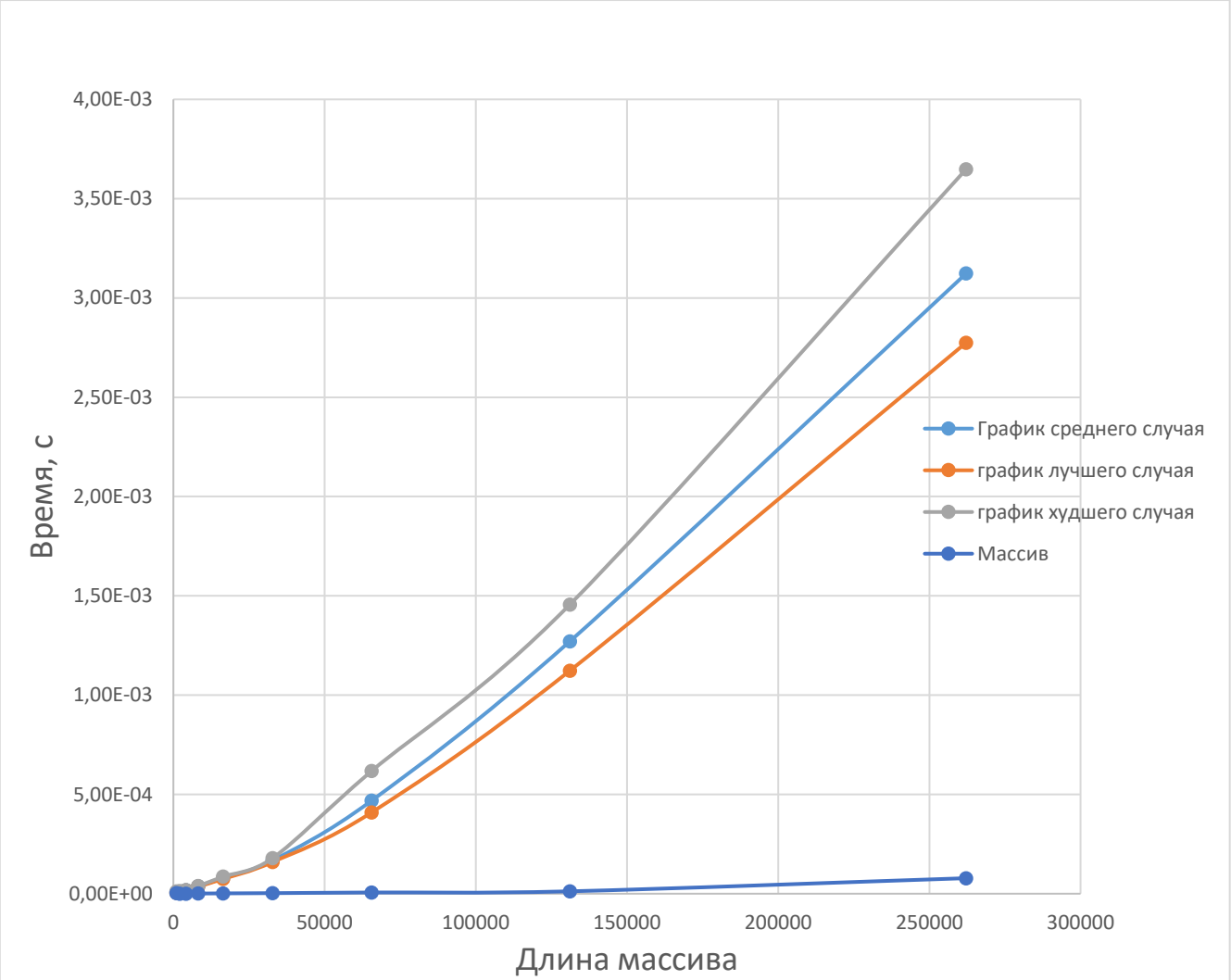

Рис.20.1. График времени, за которое ищется узел в Бинарном дереве в сравнении с сортированным массивом.

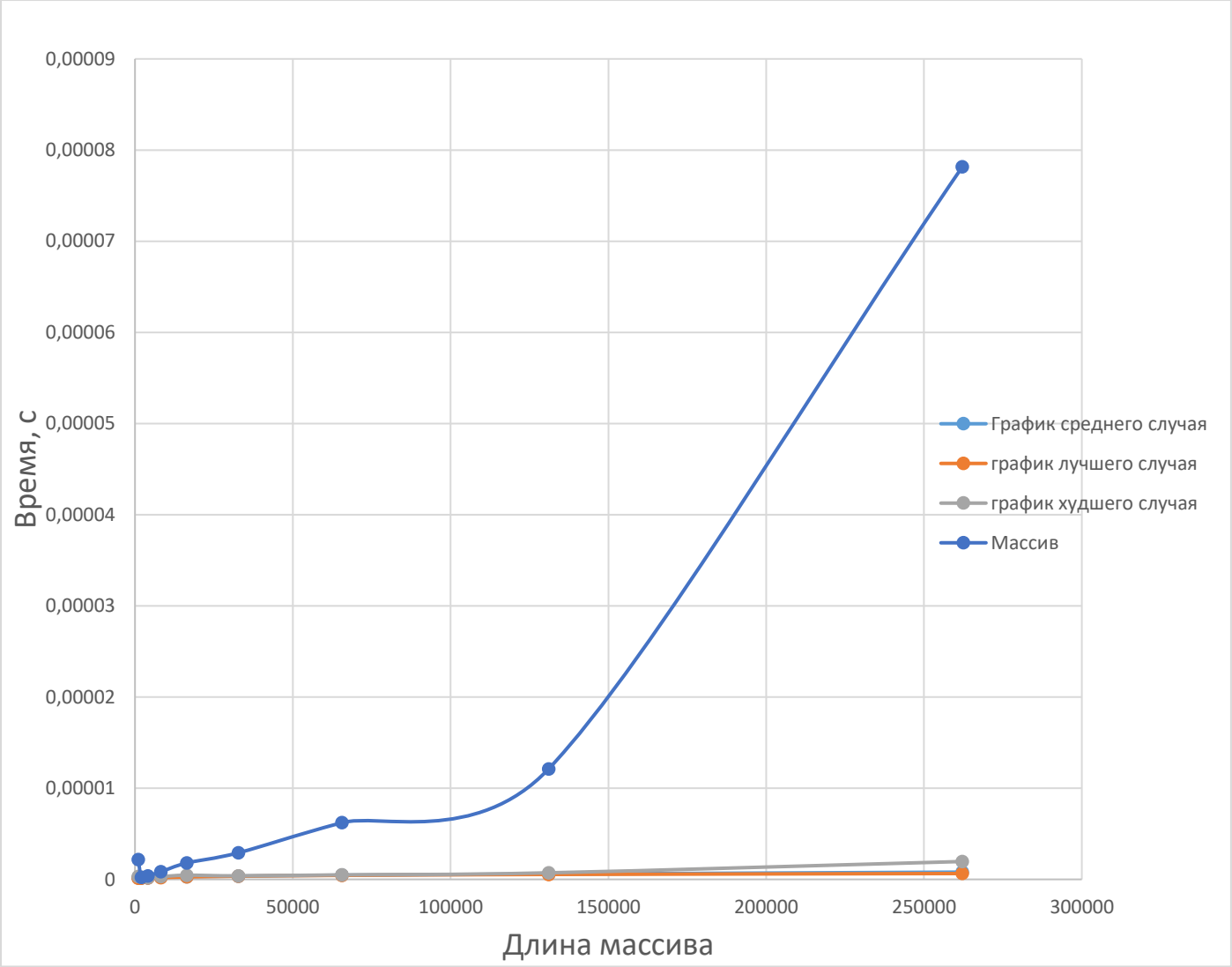

Рис.20.2. График времени, за которое ищется узел в AVL-дереве в сравнении с сортированным массивом.

По графикам видно, что поиск в бинарном и avl дереве работает во всех случаях быстрее, чем у массива стандартного, кроме бинарного сортированного дерева.

12) Замерил удаление у несортированного Бинарного дерева

Таблица 9

Время удаления одного элемента у несортированного Бинарного-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	1.6E-07	8.54E-07	2.76E-07
11	1.92E-07	2.15E-07	2.04E-07
12	2.26E-07	2.62E-07	2.42E-07
13	2.72E-07	3.35E-07	3.03E-07
14	3.64E-07	5.75E-07	4.23E-07
15	4.82E-07	6.28E-07	5.12E-07
16	5.97E-07	6.6E-07	6.24E-07
17	7.36E-07	9.23E-07	7.90E-07
18	9.34E-07	1.96E-06	1.4E-06

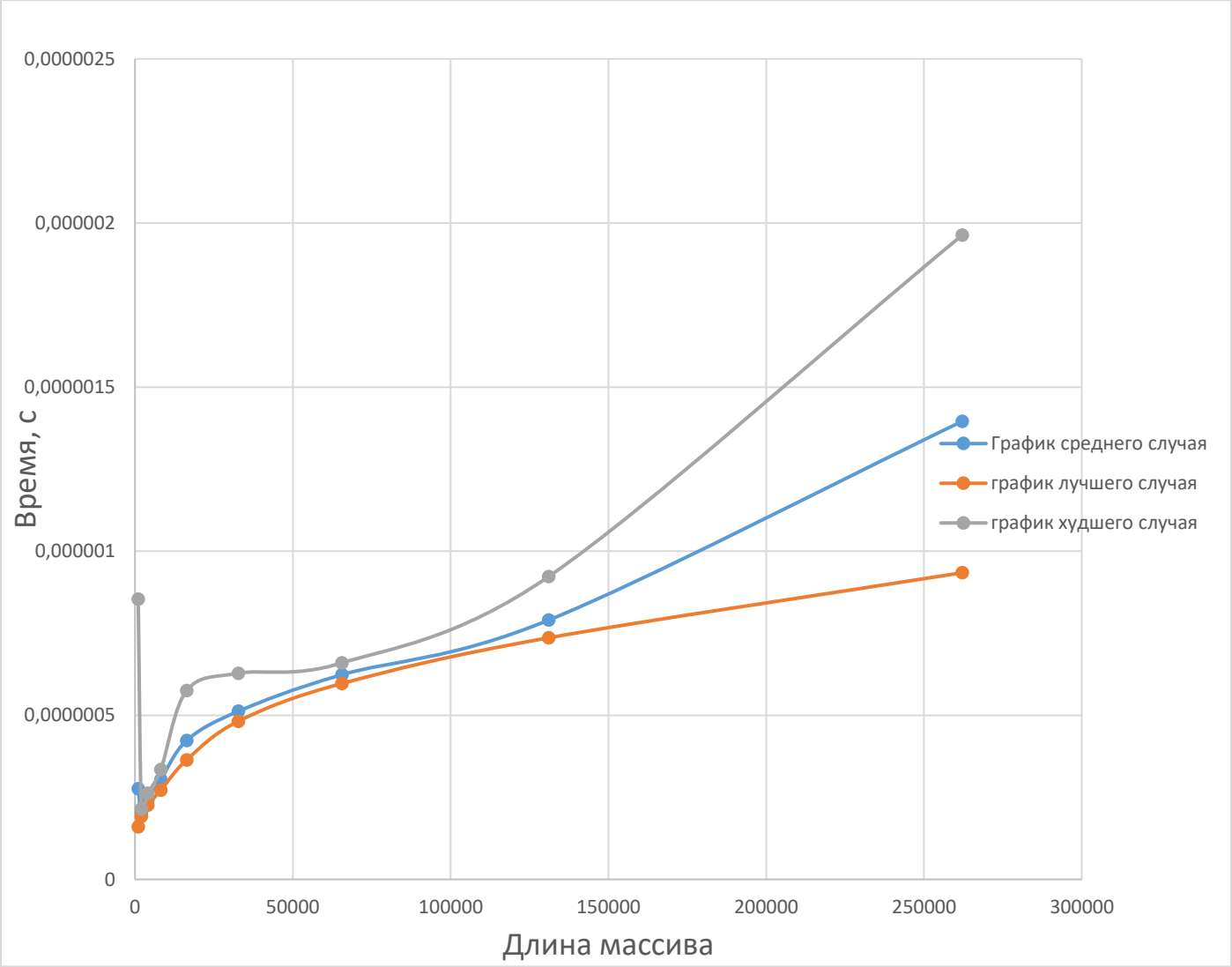

Рис.21. График времени, за которое удаляется узел в несортированном Бинарном дереве

13) Замерил удаление у сортированного Бинарного дерева

Таблица 10

Время удаления одного элемента у сортированного Бинарного-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	5.66E-06	9.97E-06	7.41E-06
11	1.57E-05	1.94E-05	1.65E-05
12	4.16E-05	4.49E-05	4.28E-05
13	9.75E-05	1.06E-04	9.98E-05
14	2.15E-04	2.57E-04	2.25E-04
15	5.12E-04	1.05E-03	6.61E-04
16	1.80E-03	2.55E-03	2.16E-03
17	4.41E-03	0.005399	4.82E-03
18	0.009104	0.01083	0.00992

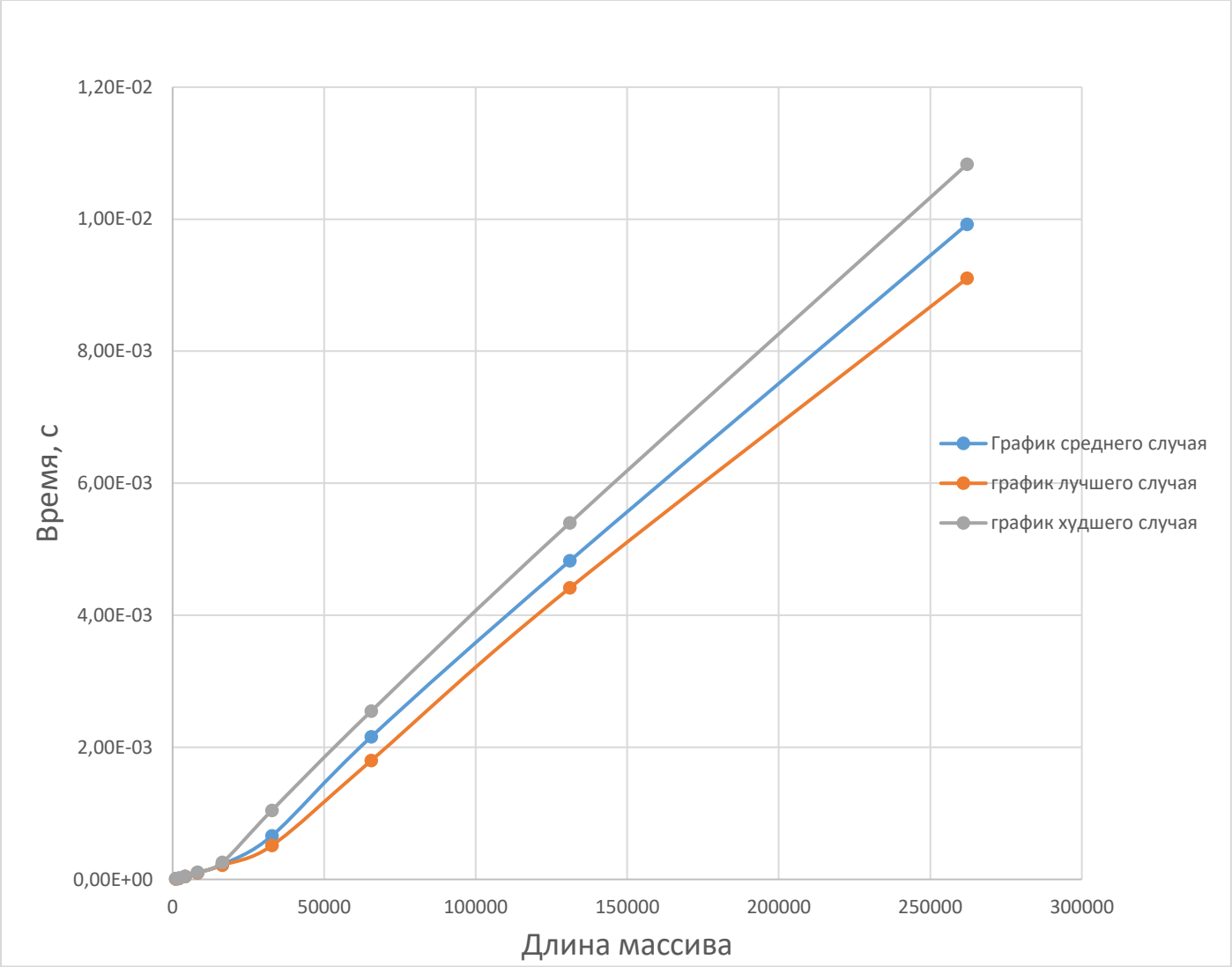

Рис.22. График времени, за которое удаляется узел в сортированном Бинарном дереве

По графикам видно, что удаление в бинарном дереве с большим кол-вом элементов работает медленнее, чем у несортированного.

14) Замерил удаление у несортированного AVL-дерева

Таблица 11

Время удаления одного элемента у несортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	1.07E-05	3.07E-05	1.58E-05
11	2.78E-05	3.86E-05	3.15E-05
12	6.78E-05	8.51E-05	7.72E-05
13	0.000171	0.000194	0.00018
14	0.000408	0.000472	0.000445
15	0.000922	0.001073	0.000974
16	0.002007	0.003146	0.002271
17	4.09E-03	0.005885	4.84E-03
18	0.010169	0.018403	0.014273

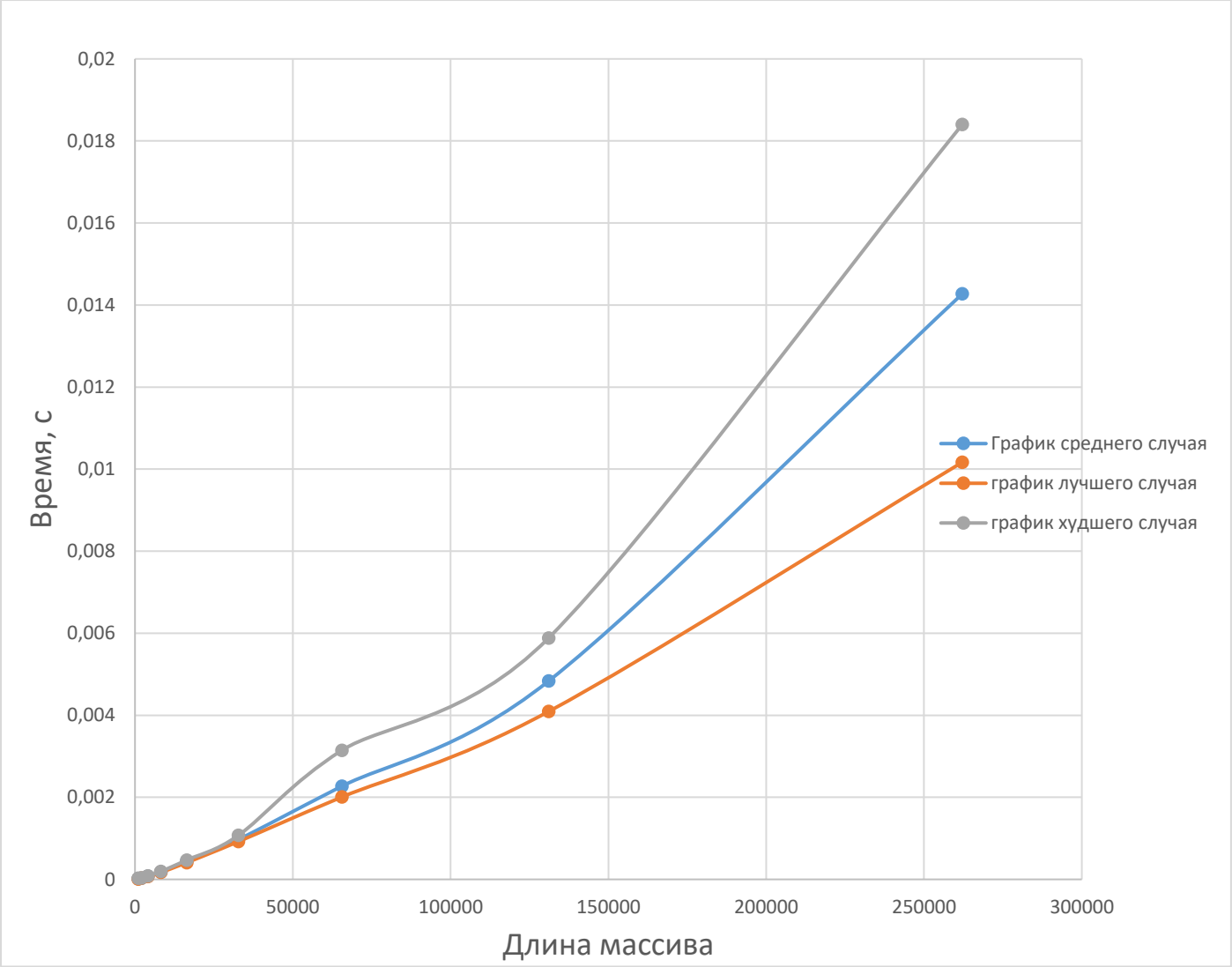

Рис.23. График времени, за которое удаляется узел в несортированном AVL-дереве

15) Замерил удаление у сортированного AVL-дерева

Таблица 12

Время удаления одного элемента у сортированного AVL-дерева в секундах

2^	Лучшее	Худшее	Среднее
10	1.69E-05	2.89E-05	2.18E-05
11	3.98E-05	5.23E-05	4.44E-05
12	8.89E-05	0.000113	0.000101
13	1.94E-04	0.000226	0.000211
14	0.000391	0.000427	0.000411
15	0.000812	0.001023	0.000881
16	0.001745	0.002108	0.001914
17	3.30E-03	0.004095	3.67E-03
18	0.008029	0.00998	0.009083

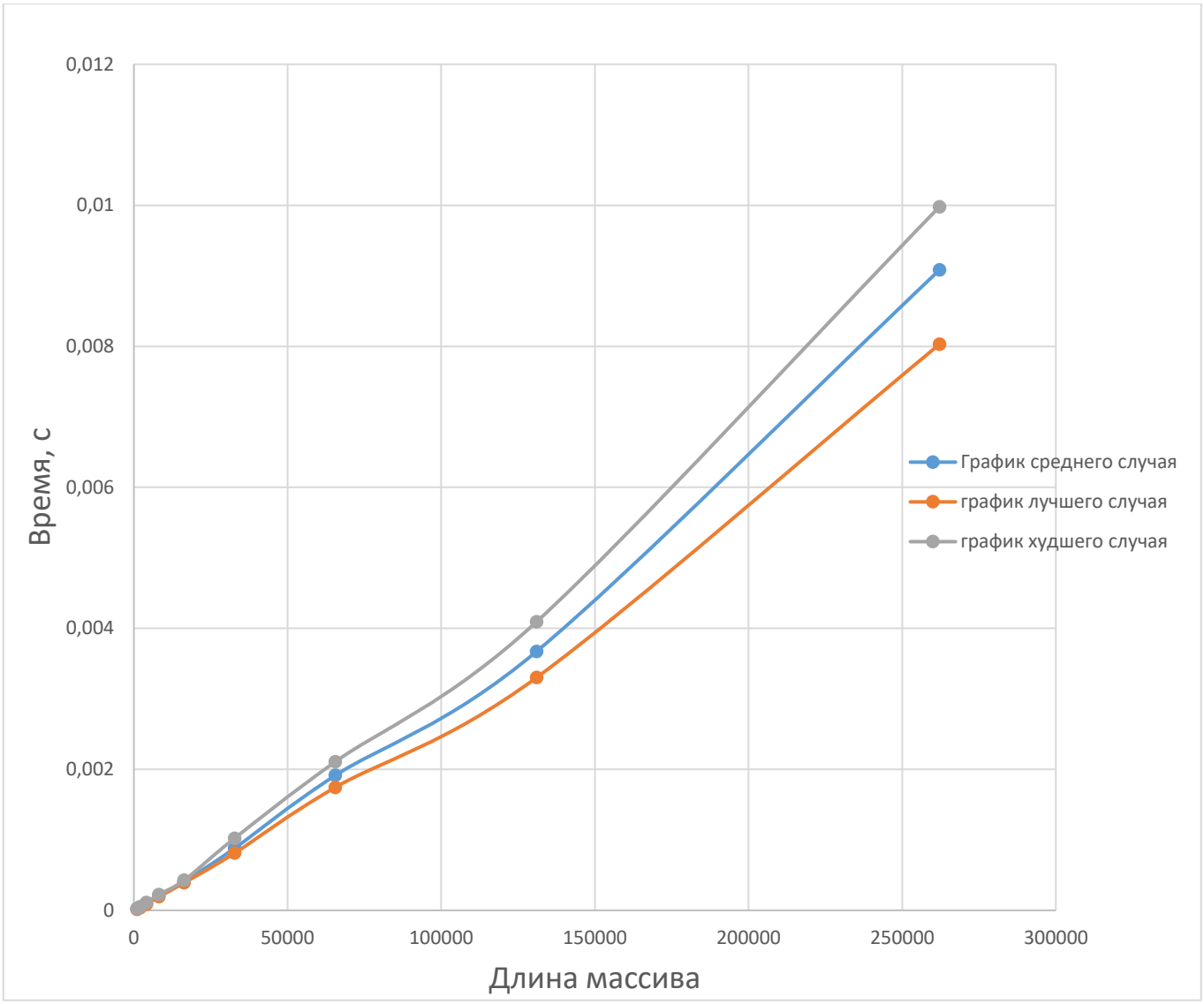

Рис.24. График времени, за которое удаляется узел в сортированном AVL-дереве

По графикам видно, что удаление в AVL-дереве с большим кол-вом элементов работает немного медленнее, чем у несортированного, но оба они медленнее, чем удаление в бинарном дереве.

Заключение.

Реализация Бинарного дерева показалась мне достаточно простой, вот уже AVL была немного сложнее. На графиках можно было увидеть, что время выполнения поиска, генерации и удаления растет вместе с количеством узлов дерева и что с заранее сортированными данными деревья работают медленнее. Также, что поиск работает быстрее у AVL-дерева.