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 № 7

Выполнил студент группы.....КС-38..... Прилепский Артем Сергеевич
Ссылка на репозиторий: .....github.com/news1d/Algorithms_and_structures

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

Дата сдачи: 14.04.2023

Оглавление

Описание задачи.

В рамках лабораторной работы необходимо изучить рандомизированное 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
- Реализовать красно черное дерево и провести все те же проверки с ним.

Описание метода/модели.

Рандомизированное дерево – это вид бинарного дерева поиска, который использует случайность при выборе операций вставки и удаления узлов. Случайный выбор элемента в качестве корня дерева обеспечивает более эффективное распределения элементов по дереву. Это позволяет достичь большего равномерного распределения элементов и, следовательно, улучшить производительность поиска.

AVL-дерево - это сбалансированное бинарное дерево поиска, в котором для каждого узла высота его двух поддеревьев отличается не более чем на 1.

При вставке или удалении элементов из AVL-дерева, если его структура нарушается и какой-либо узел становится несбалансированным (то есть его высота левого и правого поддеревьев

отличаются более чем на 1), то выполняется процедура балансировки. Балансировка в AVL-дереве производится путем поворотов поддеревьев вокруг узлов, чтобы восстановить сбалансированность.

Благодаря своей структуре, AVL-деревья обеспечивают быстрый доступ к данным и эффективное выполнение операций поиска, вставки и удаления элементов.

Выполнение задачи.

Для реализации программы был выбран язык программирования Python.

1) Класс узла для рандомизированного дерева:

```
class Node:
    def __init__(self, key):
        self.key = key
        self.size = 1
        self.right = None
        self.left = None
```

2) Класс рандомизированного дерева имеет следующий вид:

```
class RandTree:
    # Функция для поиска ершины с заданным значением
    def search(self, p, k):
        # Если дерево пустое
        if not p:
            return None

        # Если значение найдено
        if k == p.key:
            return p

        # Если значение меньше значения вершины, рекурсивно ищем в левом потомке
        if k < p.key:
            return self.search(p.left, k)
        else:
            # Если значение больше значения вершины, рекурсивно ищем в правом
потомке
            return self.search(p.right, k)

    # Вспомогательная функция для получения размера узла
    def _getsize(self, p):
        if not p:
            return 0
        return p.size

    # Вспомогательная функция для установления корректного размера дерева
    def _fixsize(self, p):
        p.size = self._getsize(p.left) + self._getsize(p.right) + 1

    # Функция для правого поворота
    def _rotateright(self, p):
        # Новой вершине присваивается значение левого потомка родительской
вершины
        q = p.left
        # Левому потомку родительской вершины присваивается значение правого
потомка новой вершины
        p.left = q.right
        # Правому потомку новой вершины присваивается значение родительской
вершиной
        q.right = p
```

```
        # Пересчитываем размеры узлов
        q.size = p.size
        self._fixsize(p)
        return q

    def _rotateleft(self, q):  # левый поворот вокруг узла q
        # Новой вершине присваивается значение левого потомка родительской
вершины
        p = q.right
        # Правому потомку родительской вершины присваивается значение левого
потомка новой вершины
        q.right = p.left
        # Левому потомку новой вершины присваивается значение родительской
вершиной
        p.left = q

        # Пересчитываем размеры узлов
        p.size = q.size
        self._fixsize(q)
        return p

    # Вспомогательная функция для вставки вершины
    def _insertroot(self, p, k):
        # Если дерево пустое, то добавляем вершину в качестве корня
        if not p:
            return Node(k)
        # Если родительская вершина больше добавляемого значения
        if k < p.key:
            p.left = self._insertroot(p.left, k)
            return self._rotateright(p)
        else:
            p.right = self._insertroot(p.right, k)
            return self._rotateleft(p)

    # Функция рандомизированной вставки новой вершины
    def insert(self, p, k):
        # Если дерево пустое, то добавляем вершину в качестве корня
        if not p:
            return Node(k)

        # С вероятностью 1/(p.size+1) добавляем вершину в корень дерева
        if random.randint(0, p.size) == 0:
            return self._insertroot(p, k)
        # Если родительская вершина больше добавляемого значения
        if p.key > k:
            p.left = self.insert(p.left, k)
        else:
            # Если родительская вершина меньше добавляемого значения
            p.right = self.insert(p.right, k)

        # Пересчитываем размеры узлов
        self._fixsize(p)
        return p

    # Вспомогательная функция объединения двух деревьев
    def _join(self, p, q):
        # Если один из корней отсутвует, то возвращаем другой
        if not p:
            return q
        if not q:
            return p

        # Если случайное число меньше размера первого дерева, то оно становится
корнем нового дерева
        if random.randint(0, p.size + q.size) < p.size:
            p.right = self._join(p.right, q)
            # Пересчитываем размер дерева
```

```
            self._fixsize(p)
            return p
        else:
            # Если случайное число больше размера первого дерева, то корнем
нового дерева становится второе дерево
            q.left = self._join(p, q.left)
            # Пересчитываем размер дерева
            self._fixsize(q)
            return q

    # Функция для удаления вершины
    def delete(self,p, k):
        if not p:
            return p
        # Если значение вершины равно искомому значению
        if p.key == k:
            q = self._join(p.left, p.right)
            return q
        # Если значение вершины больше искомого значения
        elif k < p.key:
            # Вызываем функцию для левого потомка
            p.left = self.delete(p.left, k)
        else:
            # Если значение вершины меньше искомого значения
            # Вызываем функцию для правого потомка
            p.right = self.delete(p.right, k)
        return p

    # Функция для вычисления максимальной глубины дерева
    def max_depth(self, node):
        if not node:
            return 0
        # Вызываем функцию для левого потомка
        left_depth = self.max_depth(node.left)
        # Вызываем функцию для правого потомка
        right_depth = self.max_depth(node.right)
        # Возвращаем максимальную глубину из глубин левого и правого поддеревьев,
увеличенную на 1 (текущий уровень)
        return max(left_depth, right_depth) + 1

    # Функция для вычисления глубин всех веток дерева
    def all_depths(self, node):
        if node is None:
            return []
        else:
            # Создаем пустой список, в который будем добавлять глубины потомков
            depths = []
            # Для левого и правого потомка
            for child in [node.left, node.right]:
                # Рекурсивно вызываем функцию, чтобы получить список глубин его
потомков
                child_depths = self.all_depths(child)
                # Каждую глубину из полученного списка добавляем в основной
список
                for depth in child_depths:
                    depths.append(depth + 1)
            # Если список пустой, то добавляем 1, т.к. корень существует
            if len(depths) == 0:
                depths.append(1)
            return depths
```

3) Класс узла для AVL дерева. Параметр height отвечает за высоту:

```
class AVLNode:
    def __init__(self, value):
        self.value = value
```

```
        self.left = None
        self.right = None
        self.height = 1
```

4) Реализованный класс для AVL дерева:

```
class AVLTree:
    def __init__(self):
        self.root = None

    # Вспомогательная функция для получения высоты узла
    def  get height(self, node):
        if node is None:
            return 0
        return node.height

    # Вспомогательная функция для получения баланс-фактора узла
    def _get_balance_factor(self, node):
        if node is None:
            return 0
        return self._get_height(node.left) - self._get_height(node.right)

    # Получение узла с минимальным значением из дерева
    def _get_min_value_node(self, node):
        if node is None or node.left is None:
            return node
        return self._get_min_value_node(node.left)

    # Функция для правого поворота
    def _right_rotate(self, node):
        # Новой вершине присваивается значение левого потомка родительской
вершины
        new_root = node.left
        # Левому потомку родительской вершины присваивается значение правого
потомка новой вершины
        node.left = new_root.right
        # Правому потомку новой вершины присваивается значение родительской
вершиной
        new_root.right = node

        # Пересчитываем высоты узлов
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        new_root.height = 1 + max(self._get_height(new_root.left),
self._get_height(new_root.right))

        return new_root

    # Функция для левого поворота
    def _left_rotate(self, node):
        # Новой вершине присваивается значение левого потомка родительской
вершины
        new_root = node.right
        # Правому потомку родительской вершины присваивается значение левого
потомка новой вершины
        node.right = new_root.left
        # Левому потомку новой вершины присваивается значение родительской
вершиной
        new_root.left = node

        # Пересчитываем высоты узлов
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))
        new_root.height = 1 + max(self._get_height(new_root.left),
self._get_height(new_root.right))

        return new_root
```

```
    # Функция для вставки вершины
    def _insert_node(self, node, value):
        # Если дерево пустое, то добавляем вершину в качестве корня
        if node is None:
            return AVLNode(value)
        # Если родительская вершина больше добавляемого значения
        elif value < node.value:
            node.left = self._insert_node(node.left, value)
        else:
            node.right = self._insert_node(node.right, value)

        # Обновляем высоту текущей вершины
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        # Получаем баланс-фактор текущей вершины
        balance_factor = self._get_balance_factor(node)

        # Если вершина несбалансирована, то выполняем соответствующие операции
для балансировки дерева
        if balance_factor > 1:
            # Если баланс левого потомка неотрицательный
            if self._get_balance_factor(node.left) >= 0:
                # Правый поворот родительской вершины
                return self._right_rotate(node)
            else:
                # Левый поворот левого потомка
                node.left = self._left_rotate(node.left)
                # Правый поворот родительской вершины
                return self._right_rotate(node)
        if balance_factor < -1:
            # Если баланс правого узла неположительный
            if self._get_balance_factor(node.right) <= 0:
                # Левый поворот родительской вершины
                return self._left_rotate(node)
            else:
                # Правый поворот правого потомка
                node.right = self._right_rotate(node.right)
                # Левый поворот родительской вершины
                return self._left_rotate(node)

        return node

    def insert(self, value):
        self.root = self._insert_node(self.root, value)

    # Функция для удаления вершины
    def _delete_node(self, node, value):
        if node is None:
            return node

        # Если значение вершины больше искомого значения
        elif value < node.value:
            # Вызываем функцию для левого потомка
            node.left = self._delete_node(node.left, value)
        # Если значение вершины меньше искомого значения
        elif value > node.value:
            # Вызываем функцию для правого потомка
            node.right = self._delete_node(node.right, value)
        # Если значение вершины равно искомому значению
        else:
            # Если вершина с одним или без потомков
            if node.left is None and node.right is None:
                node = None
                return node
            elif node.left is None:
```

```
                node = node.right
                return node
            elif node.right is None:
                node = node.left
                return node

            # Узел с двумя потомками
            temp = self._get_min_value_node(node.right)
            node.value = temp.value
            node.right = self._delete_node(node.right, temp.value)

        # Если дерево имело только одну вершину, то возвращаем его
        if node is None:
            return node

        # Обновляем высоту текущего узла
        node.height = 1 + max(self._get_height(node.left),
self._get_height(node.right))

        # Получаем баланс-фактор текущего узла
        balance_factor = self._get_balance_factor(node)

        # Если вершина несбалансирована, то выполняем соответствующие операции
для балансировки дерева
        if balance_factor > 1:
            # Если баланс левого потомка неотрицательный
            if self._get_balance_factor(node.left) >= 0:
                # Правый поворот родительской вершины
                return self._right_rotate(node)
            else:
                # Левый поворот левого потомка
                node.left = self._left_rotate(node.left)
                # Правый поворот родительской вершины
                return self._right_rotate(node)
        if balance factor < -1:
            # Если баланс правого узла неположительный
            if self._get_balance_factor(node.right) <= 0:
                # Левый поворот родительской вершины
                return self._left_rotate(node)
            else:
                # Правый поворот правого потомка
                node.right = self._right_rotate(node.right)
                # Левый поворот родительской вершины
                return self._left_rotate(node)

        return node

    def delete(self, value):
        self.root = self._delete_node(self.root, value)

    # Функция для поиска ершины с заданным значением
    def _search_node(self, node, value):
        # Если дерево пустое или значение найдено
        if node is None or node.value == value:
            return node

        # Если значение меньше значения вершины, рекурсивно ищем в левом потомке
        if value < node.value:
            return self._search_node(node.left, value)
        else:
            # Если значение больше значения вершины, рекурсивно ищем в правом
потомке
            return self._search_node(node.right, value)

    def search(self, value):
        return self._search_node(self.root, value)
```

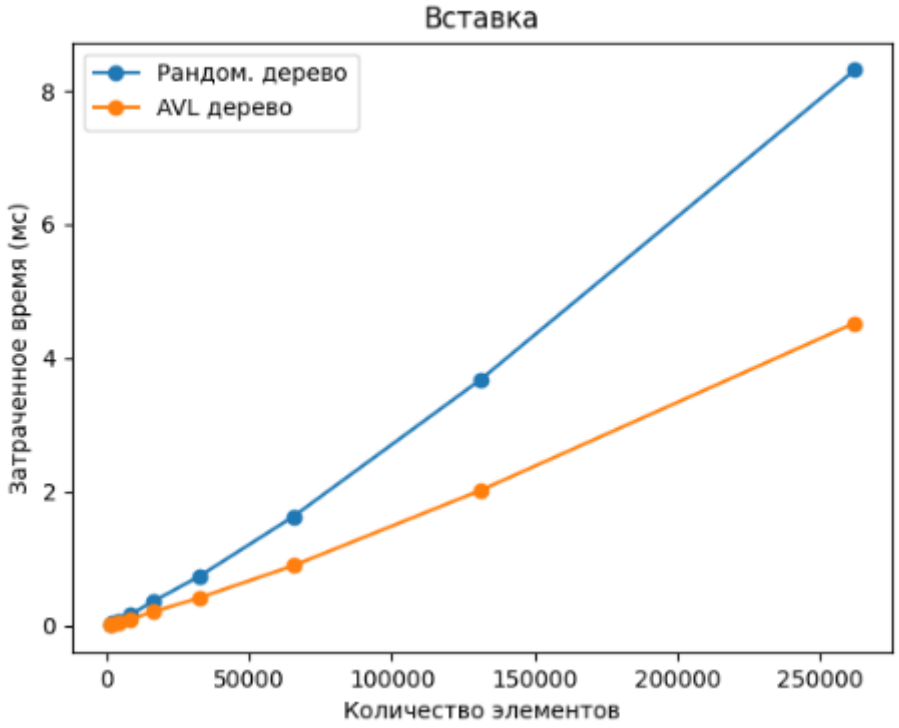


График зависимости среднего времени вставки от количества элементов

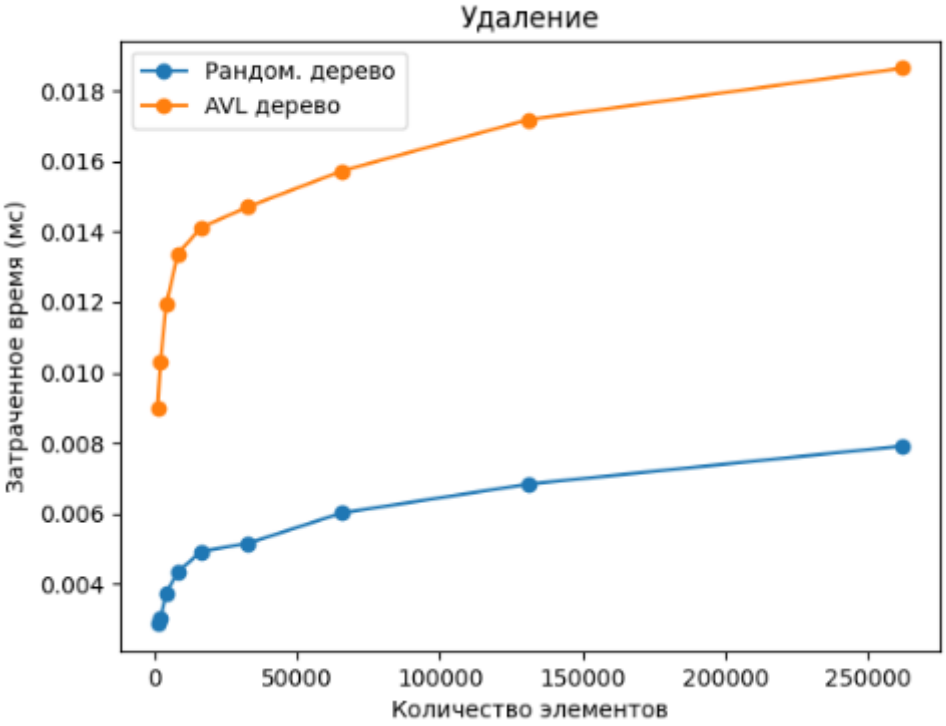


График зависимости среднего времени удаления от количества элементов

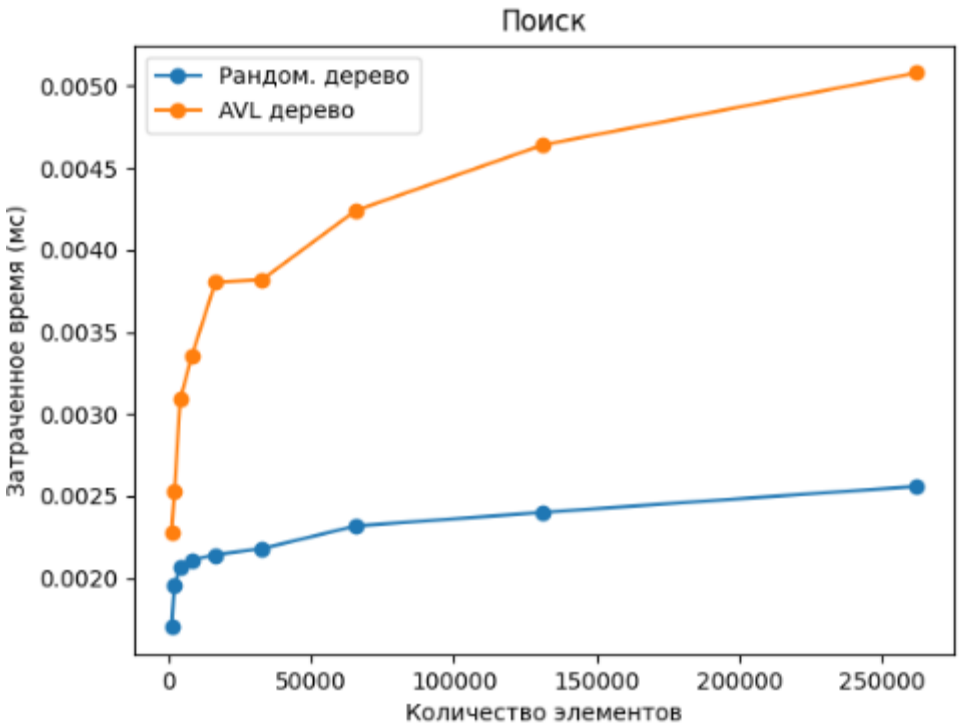


График зависимости среднего времени поиска от количества элементов

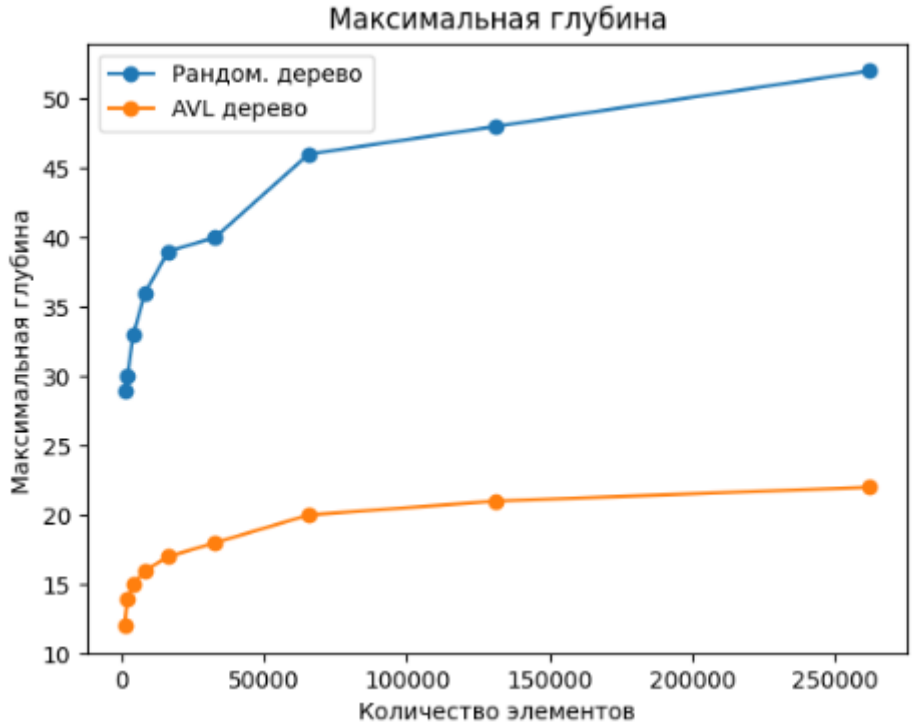
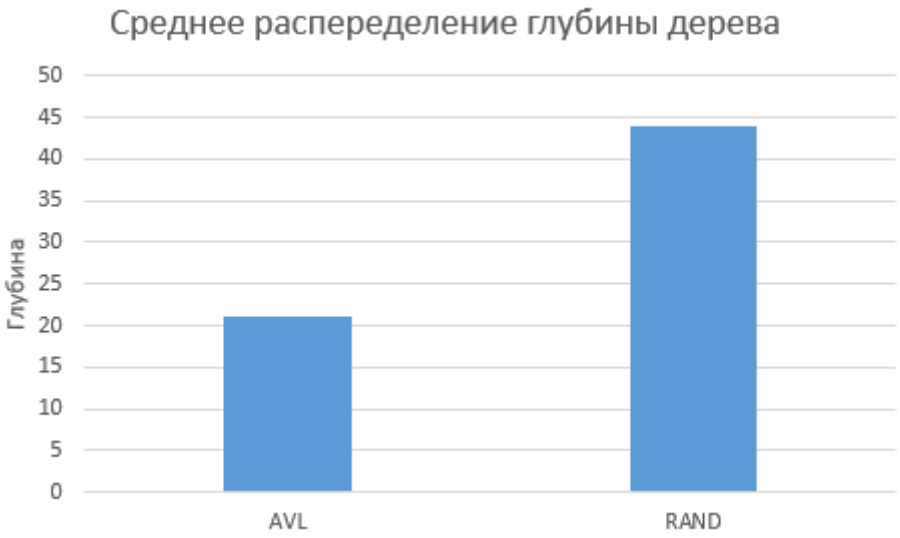
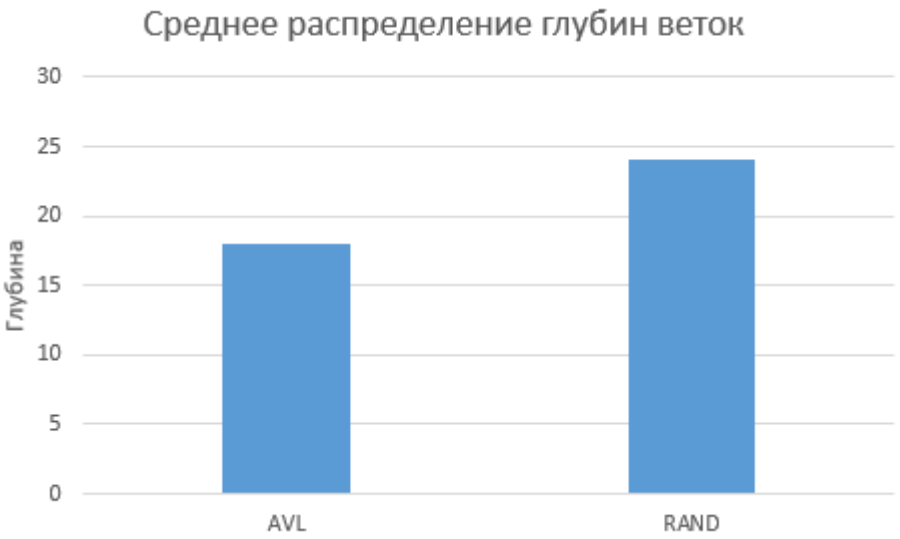


График максимальной высоты полученного дерева в зависимости от количества элементов



Гистограмма среднего распределения максимальной глубины дерева



Гистограмма среднего распределения глубин веток дерева

Заключение.

Подведем итоги, лучший показатель времени вставки – AVL дерево, удаление и поиск – рандомизированное дерево. При этом средняя глубина всего дерева и отдельных веток оказалась больше у рандомизированного дерева.

Это связано с тем, что вставка элемента в AVL-дерево занимает O(log n) времени, потому что дерево постоянно перебалансируется, чтобы сохранять свой баланс. Это приводит к меньшему времени на вставку элемента в дерево по сравнению с рандомизированным деревом, где случайный баланс может привести к худшему случаю O(n) на вставку.

Однако, при поиске и удалении элементов в рандомизированном дереве, мы можем использовать свойство случайного баланса, чтобы быстро перемещаться по дереву и быстро находить и удалять элементы. В то время как AVL-деревья могут требовать большего количества

проверок и повторного балансирования, что может привести к более медленным операциям поиска и удаления.