МИНОБРНАУКИ РОССИИ

САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ «ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)

Кафедра МО ЭВМ

КУРСОВАЯ РАБОТА

по дисциплине «Алгоритмы и структуры данных»

Тема: «Исследование абстрактных структур данных. АВЛ-дерево vs Хеш-таблица (метод цепочек)»

Студент гр. 0304	_________________	Алексеев Р.В.
Преподаватель	_________________	Берленко Т.А.

Санкт-Петербург

2021

ЗАДАНИЕ

НА КУРСОВУЮ РАБОТУ

Студент Алексеев Р.В.

Группа 0304

Тема работы: Исследование абстрактных структур данных. АВЛ-дерево vs Хеш-таблица (метод цепочек)

Исходные данные:

АВЛ-дерево vs Хеш-таблица (метод цепочек). **Исследование.** "Исследование" - реализация требуемых структур данных/алгоритмов; генерация входных данных (вид входных данных определяется студентом); использование входных данных для измерения количественных характеристик структур данных, алгоритмов, действий; сравнение экспериментальных результатов с теоретическими. Вывод промежуточных данных не является строго обязательным, но должна быть возможность убедиться в корректности алгоритмов.

Содержание пояснительной записки:

«Содержание», «Введение», «Реализация структур данных», «Проверка корректности реализации структур данных», «Исследование структур данных», «Сравнение структур данных» ,«Заключение», «Список использованных источников»

Предполагаемый объем пояснительной записки:

Не менее 15 страниц.

Дата выдачи задания: 26.10.2021

Дата сдачи реферата:

Дата защиты реферата:

Студент	____________________	Алексеев Р.В.
Преподаватель	____________________	Берленко Т.А.

АННОТАЦИЯ

В данной работе было проведенно исследование абстрактных структур данных таких, как АВЛ-дерево и хеш-таблица (метод обратных цепочек). Структуры были реализованы на языке Python. Были проведены проверка корректности реализации структур, сравнение времени, требуемого для различных операций со структурами.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью работы является исследование и сравнение таких абстрактных типов данных, как АВЛ-дерево и хеш-таблица с методом цепочек.

Для выполнения цели были поставлены и решены следующие задачи:

1. Реализация структур данных

2. Проверка корректности реализации структур данных

3. Измерение времени, требуемого для выполнения определенных операций у каждой из структур данных

4. Сравнение полученных результатов измерений

5. Вывод на основе сравнения

1. РЕАЛИЗАЦИЯ СТРУКТУР ДАННЫХ

1.1. Реализация АВЛ-дерева

АВЛ-дерево — это сбалансированное двоичное дерево поиска, т. е. для каждой вершины этого дерева разницы высот левого и правого поддеревьев не превосходит 1.

Дерево поддерживает операции поиска по ключу, добавления и удаления по ключу.

Классы АВЛ-дерева и узла АВЛ-дерева

Для реализации АВЛ-дерева был создан класс узла (вершины) АВЛ-дерева - *Node*. Экземпляр классы хранит в себе значение ключа, которое передается ему при инициализации, левого и правого потомков, а также высоту дерева, с корнем в этом узле, т. е. максимальную из высот поддеревьев узла. При выводе узла в консоль выводится строка с значение ключа, а также левого и правого потомков.

Был создан класс самого АВЛ-дерева - *AVLTree*, экземпляр класса хранит кореневой узел дерева. В классе реализованы методы добавления узла, поиска узла по ключу, удаления узла по ключу, а также дополнительные методы для поулчения высоты дерева, балансировки дерева, вывода дерева в консоль, малых и больших левого и правого поворотов дерева, поиска и удаления узла с минимальных значением ключа.

Поиск узла в АВЛ-дереве по ключу

Поиск в АВЛ-дереве основан на свойстве бинарных деревьев поиска — ключ меньший ключа родительского узла находится в левом дочернем узле, а ключ больший ключа родительского узла — в правом дочернем узле. Поэтому для посика был использован цикл *while*, в котором значение искомого ключа сравнивалось с значение ключа текущого узла, если искомый ключ был больше, то новым текущим узлом становился правый дочерний узел, если искомный ключ был меньше, то текущим узлом становился левый дочерний узел, если значение искомого ключа совпадало к значением ключа узла, то это означало, что найдем искомый узел, который возвращался обратно, если у текущего узла не было детей, а совпадающий ключ не был найдет, то возвращался узел с ключом -1 и выводилось в консоль сообщение о том, что узла с искомым ключом в дереве нет. Т.к. метод посещает, нисходящие от корня вниз, узлы, то время работы метода составляет O(h), где h — высота АВЛ-дерева, следовательно время работы поиска — O(log n), где n — количество узлов в дереве.

Балансировка АВЛ-дерева

Одним из основных свойств АВЛ-дерева, отличающих его от других двоичных деревьев поиска, является балансировка узлов так, чтобы разница высот поддеревьев одного узла была не больше 1 по модулю. Для балансировки АВЛ-дерева используются 4 вида поворотов:

1. Малый левый поворот
2. Малый правый поворот
3. Большой левый поворот
4. Большой правый поворот

Малый левый поворот используется, если высота правого поддерева больше высоты левого поддерева больше чем на 1. При повороте правый дочерний узел *b* становится родительским, а родительский узел *a* левым дочерним узлом узла *b*, изначальное левое поддерево узла *b* становится правым поддеревом узла *a*. Схема поворота представлена на рис. 1.

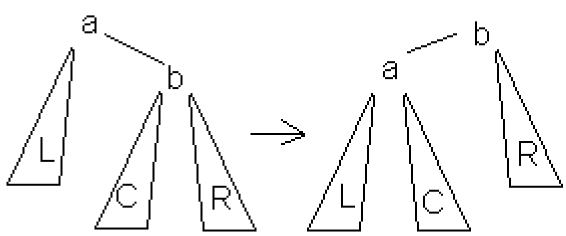


Рисунок 1 — Малый левый поворот

Малый правый поворот симметричен малому левому и используется в случае, если высота левого поддерева больше высоты правого поддерева больше чем на 1. При правом малом повороте корневой узел *a* становится правым дочерним узлом своего изначального левого дочернего узла *b,* узел *b* становится новым корневым узлом, а правое поддерево узла *b* становится левым поддеревом узла *a.* Схема поворота представлена на рис. 2.

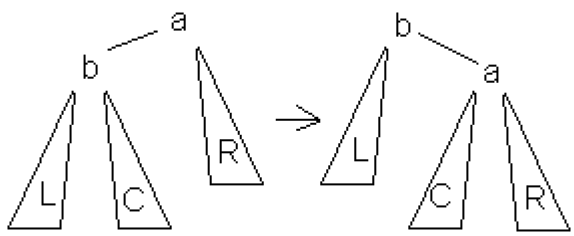


Рисунок 2 — Малый правый поворот

Большой левый поворот представляет собой последовательное выполнение малых правого и левого поворотов. Этот поворот используется в случае, если высота правого поддерева корневого узла *a* больше высоты левого поддерва *L* больше чем на 1, а также высота левого поддерева (с вершиной в узле c) правого дочернего узла *b* больше высоты правого поддерева больше чем на 1. Сначала выполняется малый правый поворот вокруг узла *b*, а потом малый левый поворот вокруг узла *a*. В итоге новым корнем дерева становится узел *c*, узлы *a* и *b* становятся соответственно левым и правым дочерними узлами. Схема бальшого левого поворота представлена на рис. 3.

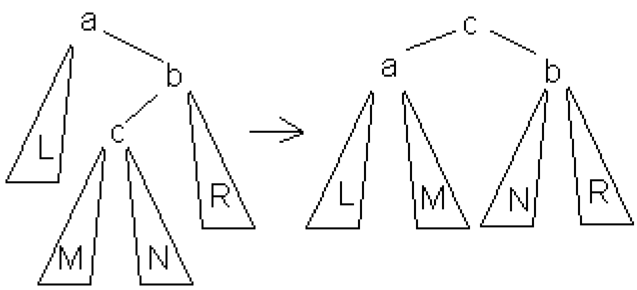


Рисунок 3 — Большой левый поворот.

Большой правый поворот симметричен большому левому. Используется в случае, если высота левого поддерева (с корнем в узле *b*) узла *a* больше чем на 1 высоты правого поддерева и высота правого поддерева узла *b* (с корнем в узле *c*) больше высоты левого поддерева больше чем на 1. В результате последовательного выполнения малого левого поворота вокруг узла *b* и малого правого поворота вокруг узла *a*, новым корневым узлом дерева становится узел *c*, а его левым и правым дочернеми узлами становятся соответственно *b* и *a*. Схема поворота представлена на рис. 4.

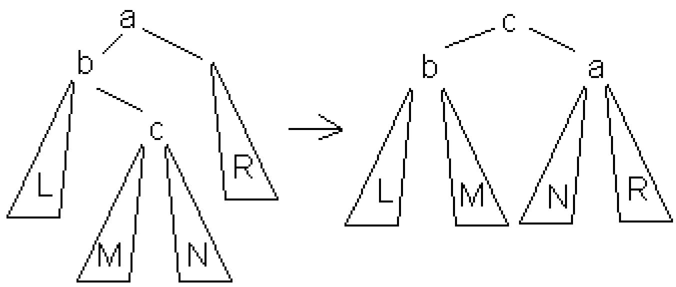


Рисунок 4 — Большой правый поворот.

Добавление узла в АВЛ-дерево

В данной реализации АВЛ-дерева добавленление узла происходит благодаря рекурсии в методе *add_key(self, key)*. Если корневой узел дерева пустой, то новый узел становится корневым, иначе вызывается метод *add_node(self, key, node)*, куда передаются новый ключ и корневой узел, метод вернет сбалансированное дерево с новым узлом. В методе значение ключа сравнивается с значением текущего узла, в соответствии с которым выбирается следующий узел для проверки, если текущий узел пустой, т. е. текущим узлом является лист, то на его место добавляется новый узел, после чего происходит балансировка дерева при каждом выходе из рекурсии, если найден узел с ключом равным новому ключу, то вставка прекращается, т. к. такой ключ уже есть в дереве. Т.к. при вставке как и при поиске метод проходит по нисходящей последовательности узлов от корня вниз, то сложность операции вставки равна O(log n), т. к. балансировка выполняется за время O(1).

Удаление узла из АВЛ-дерева по ключу

Удаление узла из АВЛ-дерева в данной реализации осуществляется благодаря рекурсии в методе *delete_key(self, key)*. Метод вызывает метод поиска узла для проверки наличия узла с требуемым ключом в дереве, если такой узел есть, то вызывается рекурсивный метод *delete_node(self, key, node)*, в который передается корневой узел дерева. В методе сравнивается значение искомого ключа с значением ключа текущего узла, в зависмости от результата сравнения либо вызывается метод удаления для правого дочернего узла, либо для левого, либо, если значение ключей совпало, происходит удаление узла. Для корректного удаления узла сначала сохраняются оба дочерних узла удаляемого узла. Если узел не имеет правого ребенка, то метод возвращет левого, в ином случае в правом поддереве находится узел с минимальным ключом, который заменяет удаляемый узел. После удаления узла и замены его на минимальный, дерево с корнем в минимальном узле балансируется, также при каждом выходе из рекурсии возвращается сбалансированное дерево с корнем в текущем узле. Итогом работы метода является возвращаение сбалансирвоанного дерева с удаленным узлом с требуемым ключом. Т.к. метод удаление задействует поиск узла, а балансировка дерева имеет сложность O(1), то сложность алгоритма удаления узла из АВЛ-дерева O(log n).

1.2. Реализация хеш-таблицы с методом цепочек

Хеш-функция — это функция, преобразующая входные данные в хеш. Хеш-таблица — это структура данных, хранящая пары ключ-значение, поддерживающая операции поиска, удаления и добавления элемента.

Коллизия — совпадение значений, получаемых от хеш-функции для двух разных наборов входных данных. Т.к. полностью исключить коллизии невозможно , то существуют различные способы разрешения коллизий, одним из которых является метод цепочек. Суть метода заключает в добавлении новых ключей с одинаковым хешем в связный список, хранящийся в одной ячейке хеш-таблицы.

Классы хеш-таблицы с методом цепочек

Для реализации хеш-таблицы с методом цепочек создан класс элемента связного списка (цепочки) — *Node.* Класс имеет поля для хранения ключа — *key*, и родителя с ребенком — *parent* и *child*. При выводе элемента в консоль выводится последовательно ключ, ключ родителя и ключ ребенка.

Также создан класс самой хеш-таблицы — *Hash_table,* хранящий простое число *P* и список длины *P*. Цепочка представляет из себя двусвязный список, т. е. элементы цепочки хранят родителя и ребенка.

Хеш-функция (хеширование)

Хеширование ключа, т. е. получение хеша, происходит в методе *hash_func(self, key),* который возвращает остаток от деления на простое число *P*.

Поиск элемента в хеш-теблице с методом цепочек

В данной реализации хеш-таблицы с методом цепочек поиск элемента осуществляется в методе *search_key(self, key)*. Сначала получается значение хеш-функции от полученного ключа, далее значение ключа элемента, который находитсся в списке под индексом равным значению хеш-функции, сравнивается с искомым ключом, если они совпадают, то метод возвращает текущий элемент, если не совпадают, то в цикле *while* начинается последовательное прохождение по цепочке из элементов с одинаковым хешем, значение ключа каждого элемента сравнивается с искомым ключом, в случае совпадаения элемент возвращается, если элементы закончились, а совпадения так и не произашло, то возвращается пустой элемент и выводится сообзение о том, что элемента с искомым ключом в хеш-таблице нет. Т.к. поиск элемента в списке осуществляется по индексу, то сложность поиска O(1), но т. к. из-за возможных коллизий искомый элемент может быть часть цепочки, то сложность поиска становится O(1) в лучшем случае и O(1+*a*), где *a* — коэффициент заполняемости таблицы, $a = \frac{n}{P}$, т. е. количество сохраненных элементов деленное на размер списка.

Добавление элемента в хеш-таблицу с методом цепочек

В данной реализации хеш-таблицы с методом цепочек добавление элемента происходит в методе *add_key(self, key). С*начала получается значение хеш-функции от нового ключа, а дальше проверяется элемент списка под индексом равным значению хеша, если там находится пустой элемент, то он заменяется новым элементов с новым ключом. В случае, если под этим индексом уже находится какой-то элемент, то в цикле *while* начинается последовательная переборка всех элементов в цепочке. Значение нового ключа сравнивается с значениями ключей элементов в цепочке, если будет найден идентичный ключ, то метод завершает работу, т. к. элемент с таким ключом уже есть в хеш-таблице. Если в процессе прохода по цепочке не был найден идентичный ключ, то новый элемент добавляется в конец цепочки, при этом у последнего элемента новый элемент сохранятесе как ребенок, а у нового элепента последний элемент сохраняется как родитель, тем самым сохраняется целостность цепочки. Т.к. доступ к элементам списка просиходит по индексу, то в лучшем случае сложность добавления нового элемента — O(1), но т. к. возможны коллизии, из-за которых придется просматривать все элементы цепочки, то в среднем сложность — O(1+*a*), где *a* — коэффициент заполняемости таблицы.

Удаление элемента из хеш-таблицы с методом цепочек.

Удаление элемента из хеш-таблицы в данной реализации происходит при помощи цикла в методе *delete_key(self, key)*. В методе сначала находится значение хеша для искомого ключа. Далее вызывается метод поска элеемнта по ключу, если элемента с искомым ключом в списке нет, то метод завершает работу. Если элемент с искомым ключом был найден, то у его родителя ребенок заменяется на ребенка удаляемого элемента, а сам элемент становится равным None. Т.к. для удаления используется поиск элемента, а само удаление из цепочки не требует циклов, то средняя сложность удаления элемента из хеш-таблицы с методом цепочек — O(1+*a*), где *a* — коэффициент заполняемости таблицы.

2. ПРОВЕРКА КОРРЕКТНОСТИ РЕАЛИЗАЦИИ СТРУКТУР ДАННЫХ

2.1. Проверка корректности реализации АВЛ-дерева

Проверка корректности реализации АВЛ-дерева проводилась путем сравнивания получаемого АВЛ-дерева с АВЛ-деревом на сервисе https://www.cs.usfca.edu/~galles/visualization/AVLtree.html.

Создание АВЛ-дерева.

В обоих сулчаях было создано дерево из узлов с ключами 15, 42, 4, 198, 17, 64, 89. АВЛ-дерево созданное сервисом представлено на рис. 5, а созданное реаилизованным алгоритмом на рис. 6.

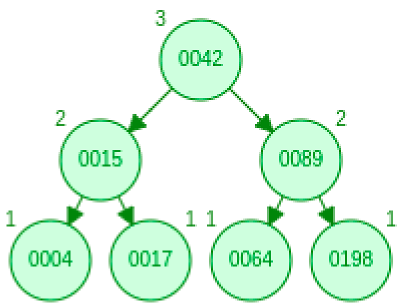


Рисунок 5 — АВЛ-дерево, созданное при помощи онлайн сервиса.

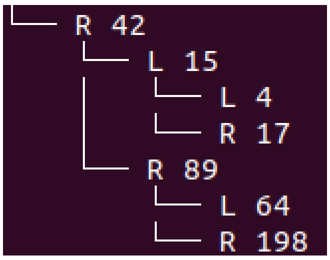


Рисунок 6 — АВЛ-дерево созданное при помощи программы.

При сравнении рисунков с результатами видно, что полученные АВЛ-деревья идентичны, следовательно создание АВЛ-дерева корректно.

Поиск узла в АВЛ-дереве

В обоих случаях был произведен поиск узла с ключом 89. Найденные узлы представленны на рис. 7 — онлайн сервис, и на рис. 8 — созданная программа.

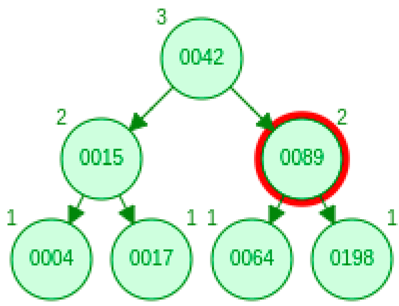


Рисунок 7 — Посик узла через онлайн сервис.

```
key: 89, left: 64, right: 198
```

Рисунок 8 — поиск узла в созданной программе.

По рисункам видно, что в обоих случаях у узла с ключом 89 левым ребенком является узел с ключом 64, а правым с ключом 198. Следовательно поиск в АВЛ-дереве корректен.

Удаление узла из АВЛ-дерева

В обоих случаях был удален узел с ключом 15. Деревья с удаленным узлом представлены на рис. 8 — онлайн сервис, на рис. 9 — программа.

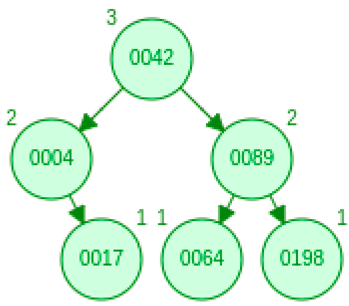
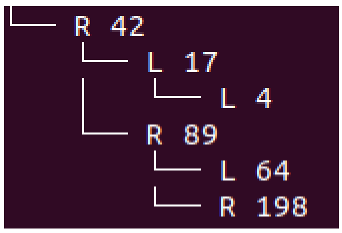


Рисунок 8 — Удаление узла через онлайн сервис.



```
└── R 42
    ├── L 17
    │   └── L 4
    └── R 89
        ├── L 64
        └── R 198
```

Рисунок 9 — Удаление узла через программу.

При сравнении рисунков видно, что узел с ключом 15 был удален в обоих случаях, а так же было выполнена балансирвока дерева. В результате получилось два немного отличающихся дерева, отличие вызвано тем, что в онлайн сервисе на место удаленного узла был помещен левый ребенок, а в программе правый. Т.к. основные свойства АВЛ-дерева не были нарушены, т. е. соблюден баланс высот, то удаление узла из АВЛ-дерева корректно.

Добавление узла в АВЛ-дерево

В оба дерев были поочередно добавлены узлы с ключами 200 и 199. Результаты добавления онлайн сервисом представлены на рис. 10, а программой на рис. 11.

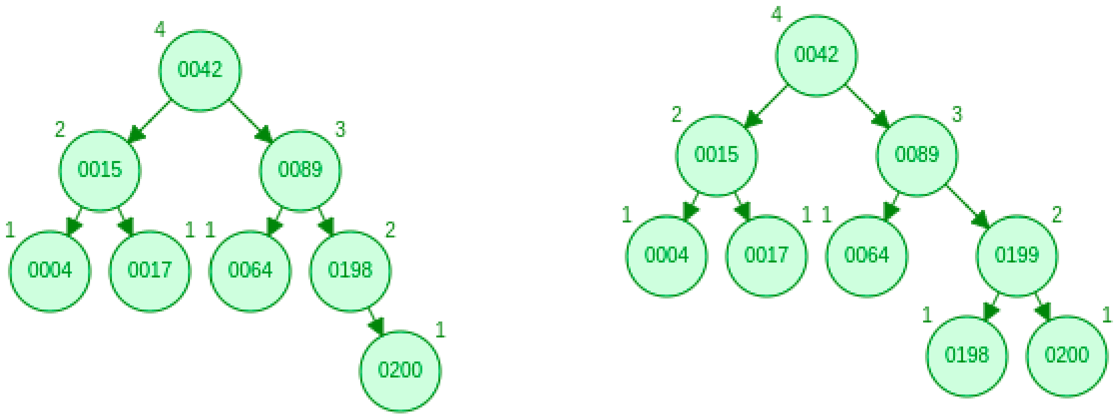


Рисунок 10 — Добавление двух узлов при помощи онлайн сервиса.

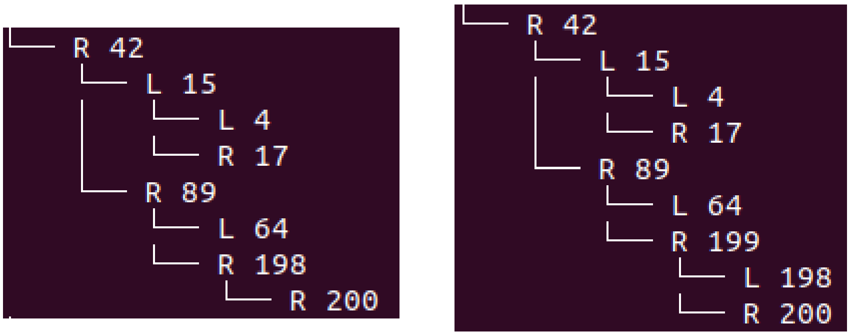


Рисунок 11 — Добавление двух узлов при помощи программы.

При сравнении полученных результаттов видно, чтов обоих случаях узел с ключом 200 был добавлен как ребенок к листу с ключом 198, а после добавления узла с ключом 199 была выполнена балансирвока дерева, в результат которой представлен на рисунках. Т.к. оба дерева после добавления узлов совпадают, то добавление узлов в АВЛ-дерево корректно.

2.2. Проверка корректности реализации хеш-таблицы с методом цепочек

Для проверки корректности реализации хеш-таблицы с методом цепочек был проведен ряд тестов. В первом тесте выводился результат хеш-функции от чисел равных 984 по модулю *P*, теоретически результат должен был совпадать. Результаты представлены на рис. 12.

```
H(984) = 984
H(8741) = 984
H(32012) = 984
```

Рисунок 12 — Значения хеш-функции от чисел равных по модулю *P*.

По рисунку видно, что теоретические результаты совпадают с практическими, следовательно хеш-функция корректна.

Для проверки добавления и поиска элементов из хеш-таблицы с методом цепочек подбирались элементы с одинкавым значение хеш-функции и последовательно добвлялись в таблицу. Т.к. в качестве метода разрешения коллизий использовался метод цепочек, то новые элементы с одинаковым хешем должны были последовательно форминровать двусвязный список. В хеш-таблицу были последовательно добавлены элементы с ключами 984, 8741, 32012, после чего был произведен сначала поиск по ключу 8741, потом добавление ключа 39769, а потом удаление ключа 8741. Результаты представленны на рис. 13.

```
Search Key: 8741, parent: 984, child: 32012
Add Key: 39769, parent: 32012, child: None
```

Рисунок 13 — Поиск и добавление ключа в хеш-таблицу.

По рисунку видно, что ключ 8741 находится в цепочке после 984 и перед 32012, т. е. в том же порядке, в котором ключи добавляли в хеш-таблицу, следовательно теоретические рассчеты совпадают с результатми. Следовательно поиск элемента корректен. Также по рисунку видно, что добавленный ключ находится в цепочке после ключа 32012, как и должно было быть по теоретическим рассчетам, следовательно добавление ключа реализованно корректно.

3. ИССЛЕДОВАНИЕ СТРУКТУР ДАННЫХ

Для исследования обоих абстрактных структур данных был написан код в файле *research.py*. В файле в отдельных функциях для поиска, добавления и удаления элементов, создается набор ключей от 0 до 99999 включительно, набор ключей одинаков для обоих структур. При добавлении каждой 1000 ключей в структуры производится соответствующая операция (поиск, добавление или удаление) с случайным ключом, лежащим в диапозоне от 0 до максимального добавленного ключа, в случае добавления ключа от 100000 до 101000.

Для замера времени на выполенение операций в методах добавления, удаления и поиска при начале работы сохраняется время, после окончания работы сохраненное время вычитается из текущего, и разница возвращается. Для удобства работы с числами время возвращается в милисекундах.

3.1. Исследование АВЛ-дерева

Поиск элемента в АВЛ-дереве

Теоретически поиск элемента в АВЛ-дереве происходит за O(log n), где n — количество элементов в дереве. В исследовании были произведены замеры времени поиска случайных элементов в зависимости от количества элементов в дереве. Результаты представлены на рисунке 14.

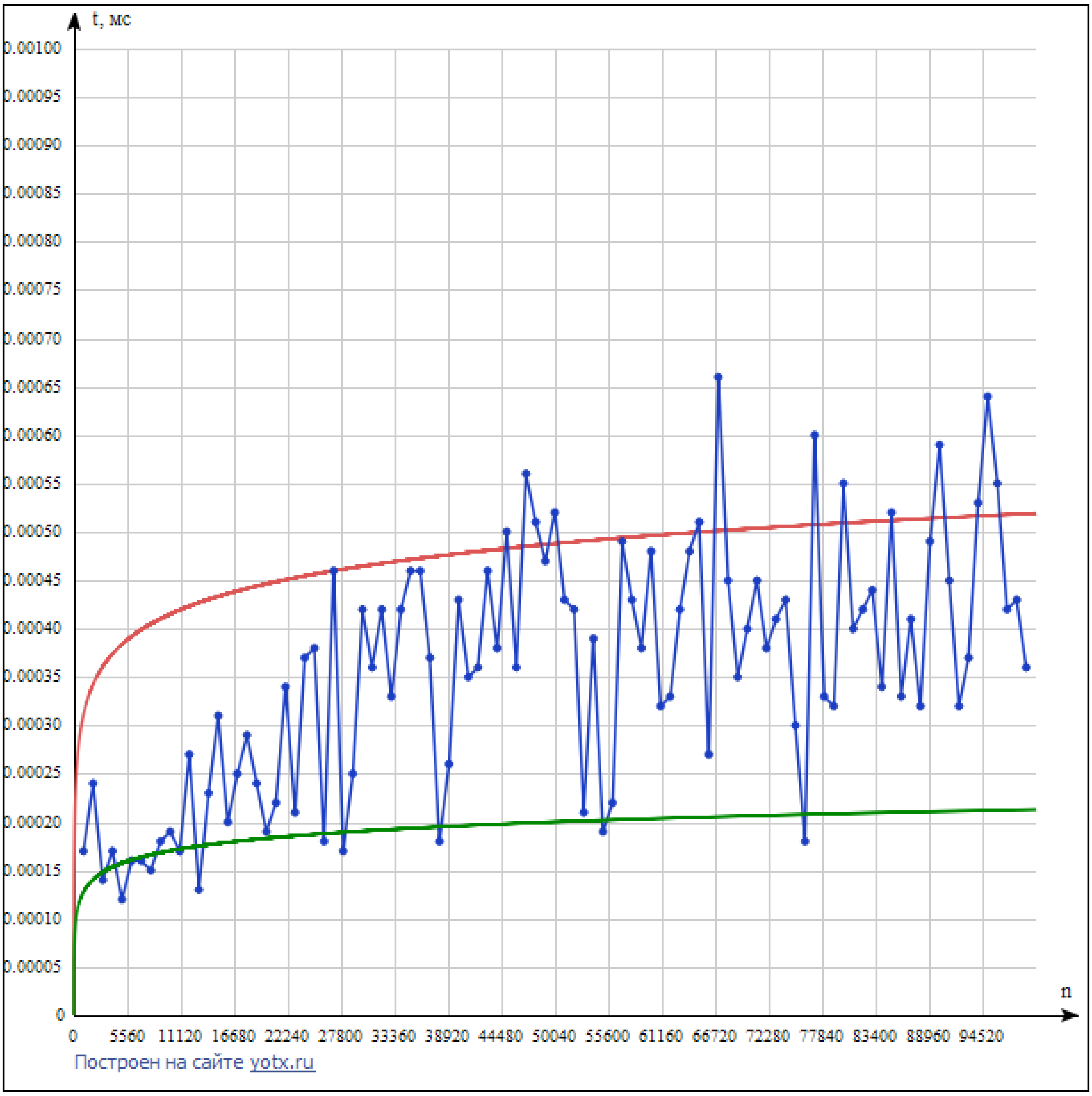


Рисунок 14 — Зависимость времени поиска элемента от количества элементов в дереве. (Синяя — зависимость времени поиска, красная — верхняя ассимптотачиеская граница, зеленая — нижняя ассмптотическая граница).

На графике прослеживается логаримическая зависимость времени поска от количества элементов в дереве, следовательно практические результаты подтверждают теоретические.

Добавление элемента в АВЛ-дерево

Теоретически добавление элемента в АВЛ-дерево происходит за O(log n). Для исследования добавления элементов в АВЛ-дерево производился замер времени добавления элемента через каждые 1000 добавлений. Результаты исследования представлены на рисунке 15.

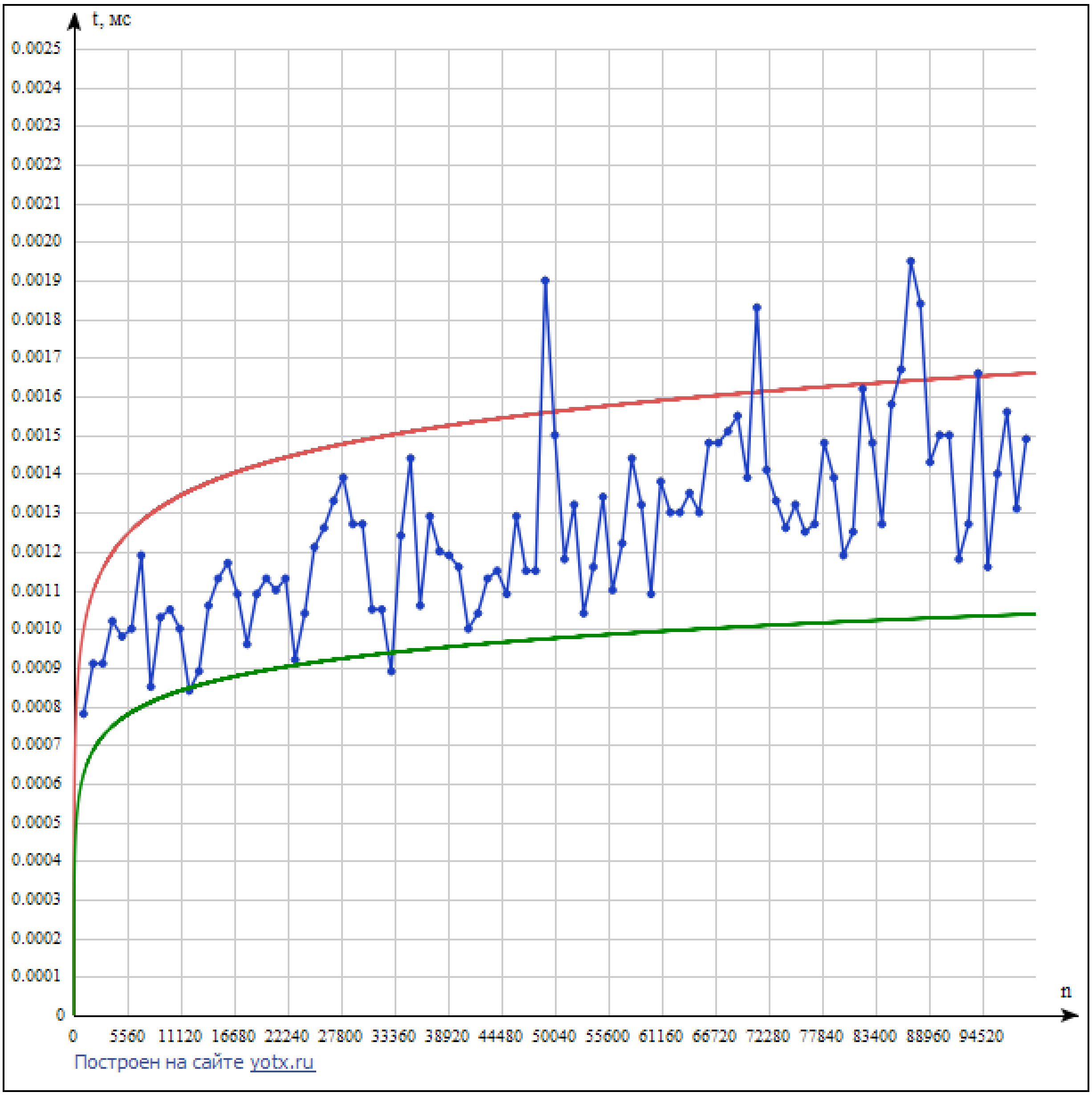


Рисунок 15 — Зависимость времени добавления элемента от количества элементов в дереве (Синяя — зависимость времени от количества элементов, красная — верхняя ассимптотачиеская граница, зеленая — нижняя ассмптотическая граница).

На графике прослеживается логаримическая зависимость времени от количества узлов в дереве, что подтверждает теорию.

Удаление элементов из АВЛ-дерева

Теоретически удаление элемента из АВЛ-дерево должно происходить за O(log n), т. к. при удалении происходит поиск элемента. Для исследования были произведены замеры времени удаления случайного элемента при добавлении каждой 1000 элементов. Результаты представлены на рисунке 16.

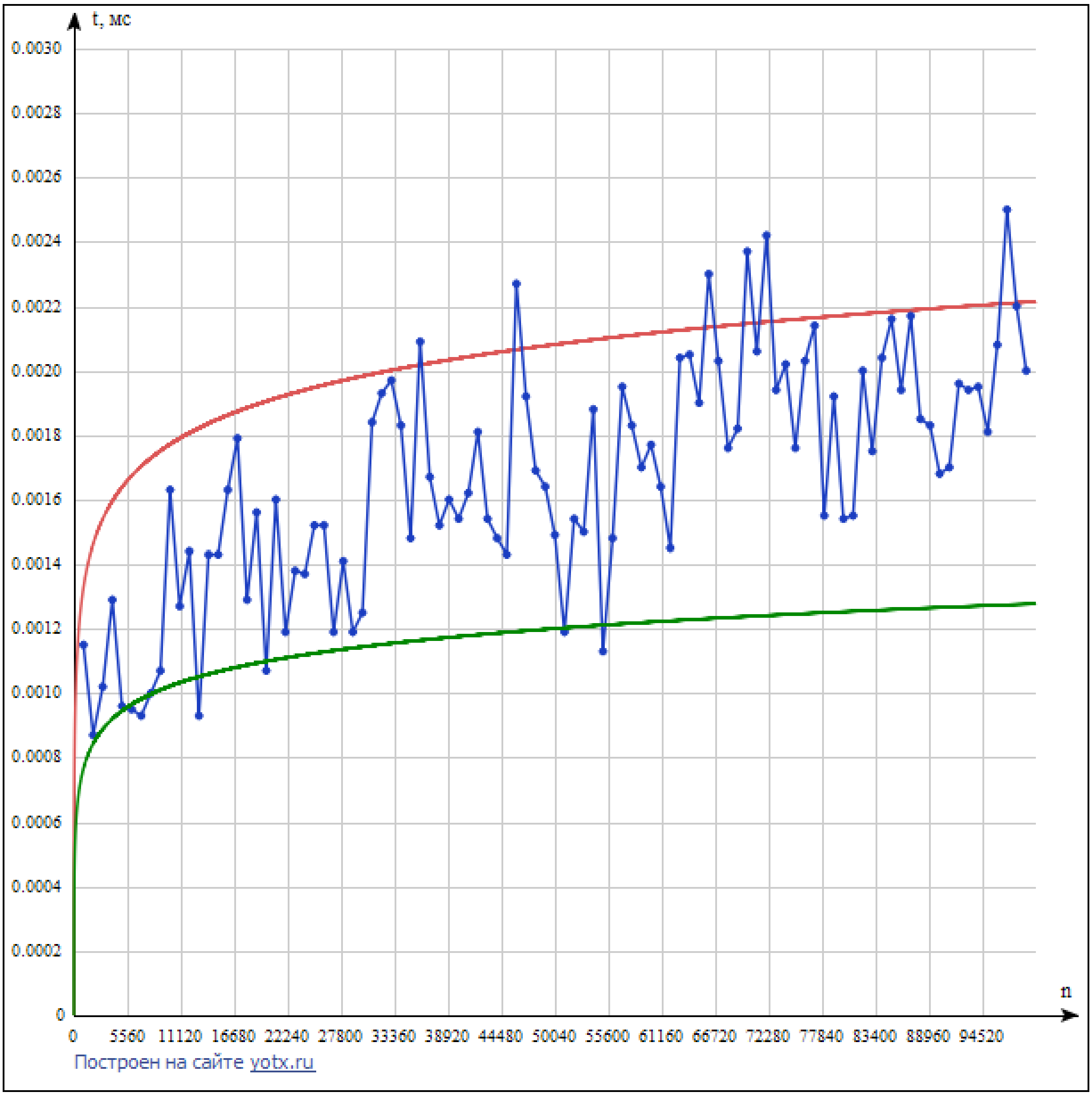


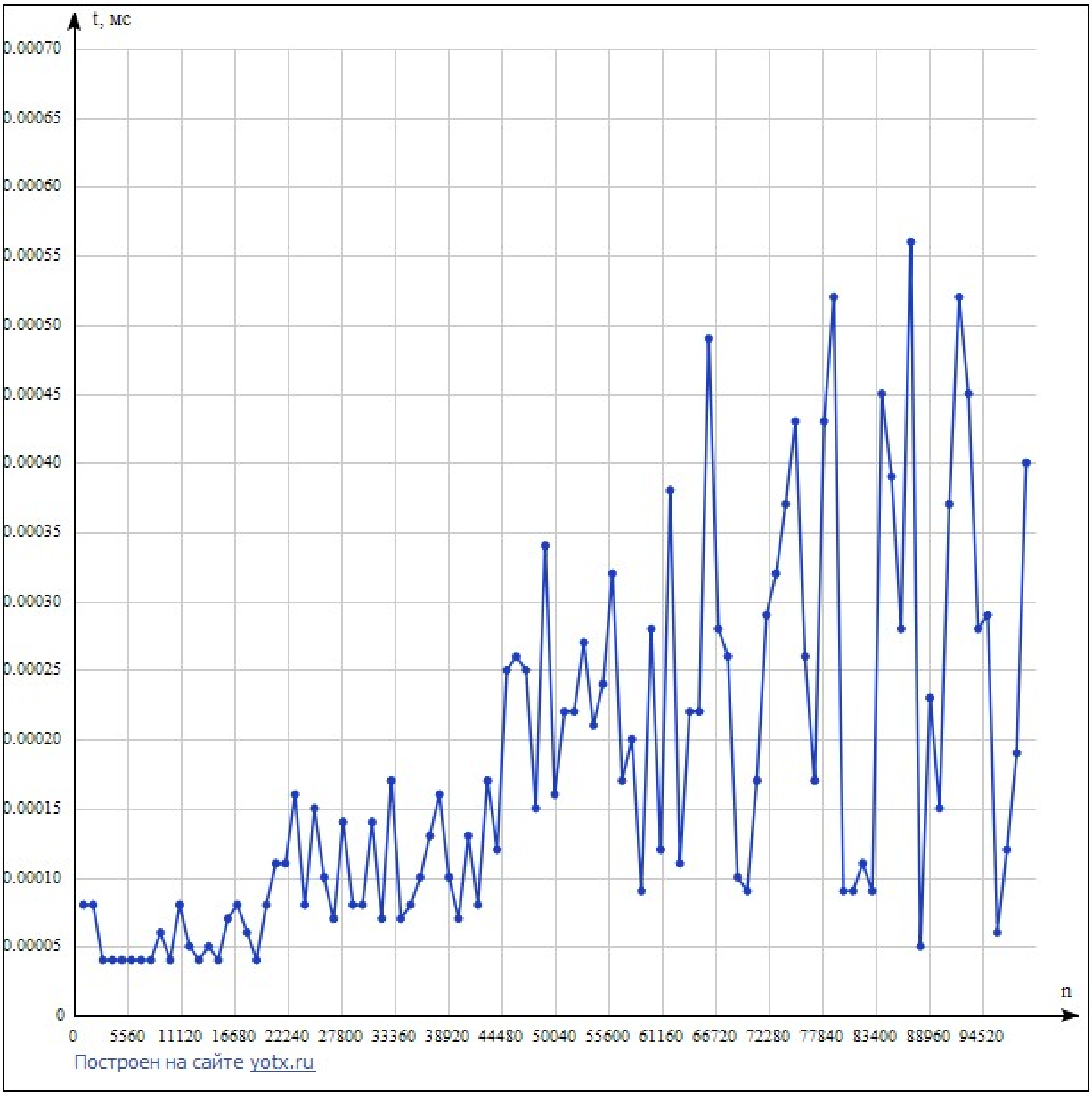
Рисунок 16 — Зависимость времени удаления от количества элементов в дереве (Синяя — зависимость времени от количества элементов, красная — верхняя ассимптотачиеская граница, зеленая — нижняя ассмптотическая граница).

На графике зависимости времени удаления элемента от количества элементов в дереве прослеживается логарифмическая зависимость, что подтверждает теоритическую оценку.

3.2. Исследование хеш-таблицы с методов цепочек

Поиск элемента в хеш-таблице с методом цепочек

Теоретически операция поиска элемента должна в среднем выполняться за O(1+*a*), где $a = \frac{n}{P}$ - коэффициент заполняемости таблицы, где n — количество элементов в таблице, P — размер таблицы, т. е. с возрастанием количества элементов в таблице время должно также возрастать. Для исследования были произведены замеры времени поиска случайного элемента при добавлении каждой 1000 новых элементов. Результаты исследования представлены на рисунке 17.



Рисунке 17 - Зависимость времени поиска от количества элементов в хеш-таблице с методом цепочек.

По графику видно, что среднее время поиска увеличивается с увеличением количества элементов в хеш-таблице, также видно, что с увеличением количества элементов увеличивается количество лучших и худших случаев, что подтверждает теоритическую оценку.

Добавление элемента в хеш-таблицу с методом цепочек

Теоретически добавление элемента должно в среднем выполняться за O(1+*a*), т. е. с увеличением количества элементов в таблице должно увеличиваться время, требуемое для добавления нового элемента. Для исследования были произведены замеры времени для добавления каждого 1000 элемента в таблицу. Результаты исследования представлены на рисунке 18.

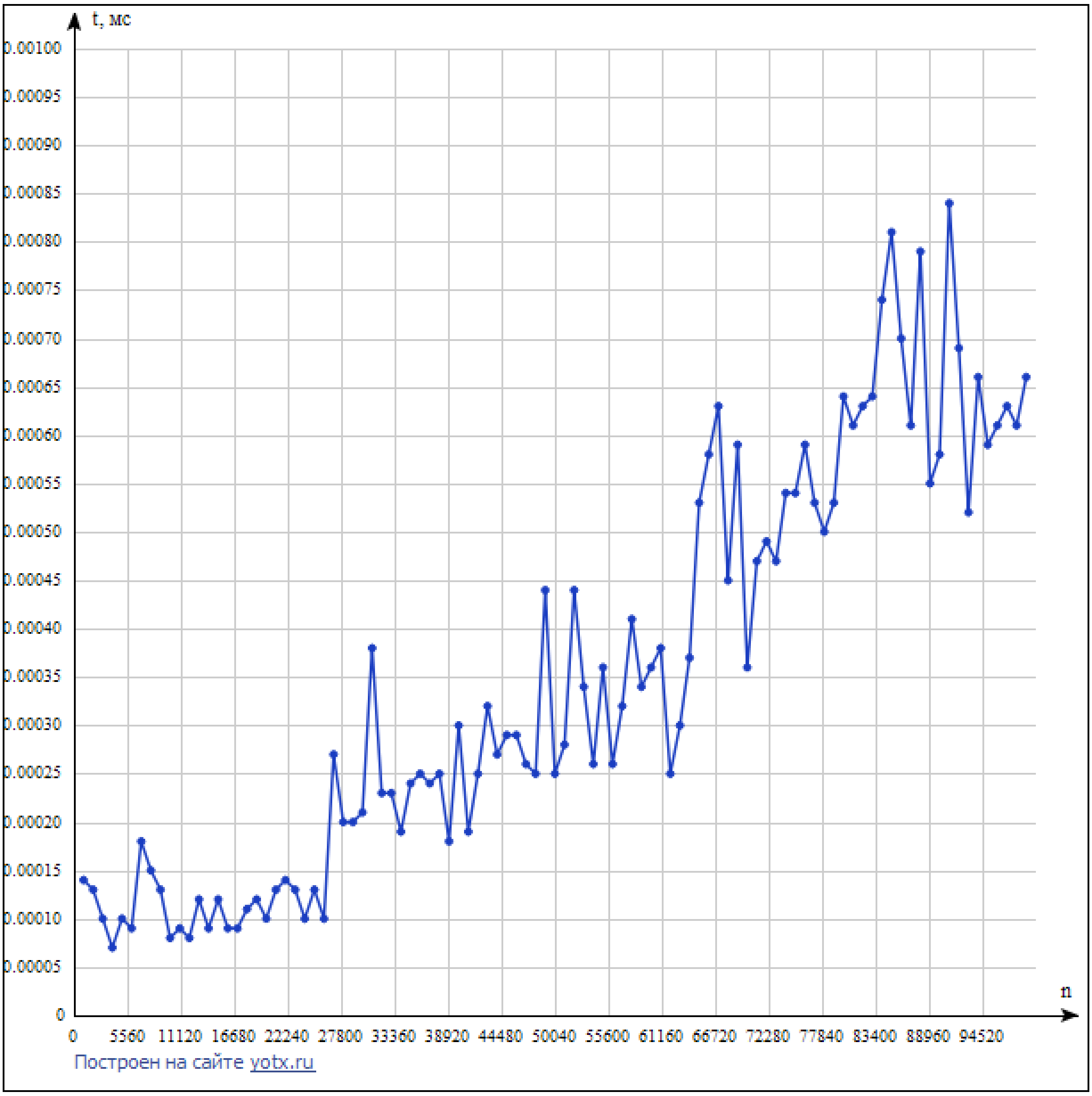


Рисунок 18 — Зависимость времени добавления элемента от количества элементов в хеш-таблице с методом цепочек.

По графику видно, что с увеличением количества элементов в хеш-таблице время, требуемое для добаления нового элемента, также увеличивается, что подтверждает теоретическую оценку.

Удаление элемента из хеш-таблицы с методом цепочек

Теоретически удаление элемента из хеш-таблицы с методом цепочек должно выполняться за O(1+*a*), т. к. для удаления элемента требуется поиск элемента,т. е. при увеличении количества элементов в таблице будет увеличиваться время удаления. Для исследования были произведены замеры времени удаления случайного элемента при добавлении каждого 1000 элемента. Результаты исследования представлены на рис. 19.

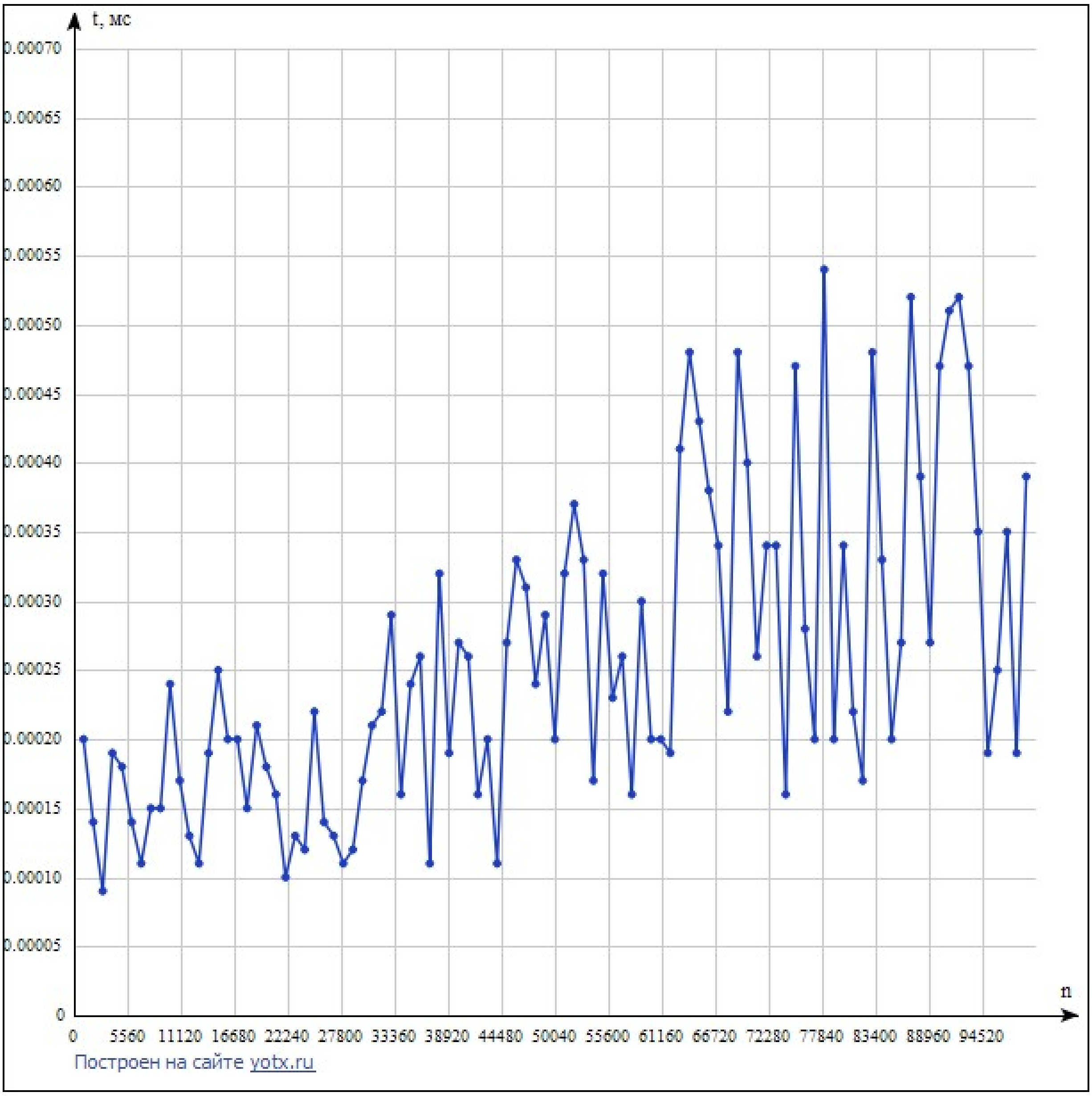


Рисунок 19 — Зависимость времени удаления элемента от количества элементов в хеш-таблцие с методом цепочек.

По граифку видно, что с увеличением количества элементов в хеш-таблице увеличивается время, требуемое для удаления элемента, что подтверждает теоретическую оценку.

4. СРАВНЕНИЕ СТРУКТУР ДАННЫХ

Сравнение поиска элементов

Были произведены замеры времени, требуемого для поиска элементов в АВЛ-дереве и в хеш-таблице с методом цепочек, при добавлении каждого 1000 элемента. Теоретически время поиска для АВЛ-дерева — O(log n), для хеш-таблицы с методом цепочек — $O(1+a)$, т. е. при малом количестве элементов поиск в хеш-таблице должен быть быстрее поиска в АВЛ-дереве, но с увеличением количества элементов время, требуемое для поиска в хеш-таблице в среднем случае, будет больше такого же времени для АВЛ-дерева. Результаты представлены на рис. 20.

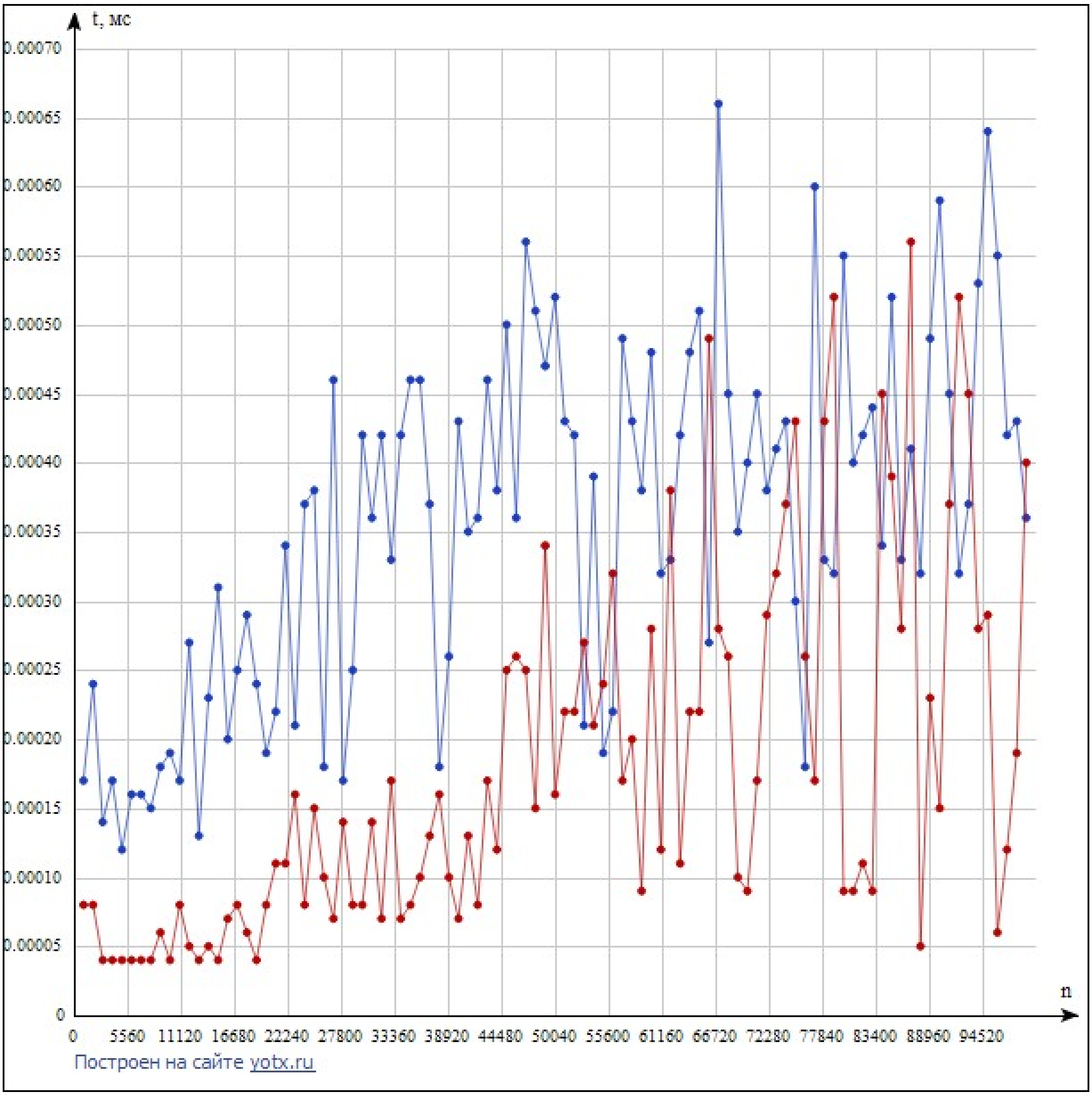


Рисунок 20 — Сравнение времени поиска в АВЛ-дереве (синий) и в хеш-таблцие с методом цепочек (красный) в зависимости от колчиества элементов.

По графику видно, что вначале время поиска в хеш-таблице было меньше времени поиска в АВЛ-дереве, но с увеличение колчиества элементов среднее время поиска элемента в хеш-таблице с методом цепочек сравнялось с временем поиска в АВЛ-дереве, что подтверждает теоретическую оценку.

Сравнение добавления элементов

Теоретически добавление нового элемента в АВЛ-дерево должно выполняться за O(log n), а для хеш-таблицы с методом цепочек за O(1+*a*). Следовательно при малом количестве элементов время добавления нового элемента в хеш-таблицу будет меньше чем время добавления в АВЛ-дерево, но с увеличением количества элементов время, требуемое для добавления элемента в хеш-таблицу с методом цепочек, будет рости бытсрее такого же времени для АВЛ-дерева. Для исспледования были произведены замеры времени, требуемого для добавления элемента, при добавлении каждого 1000 элемента. Результаты исследования представлены на рис. 21.

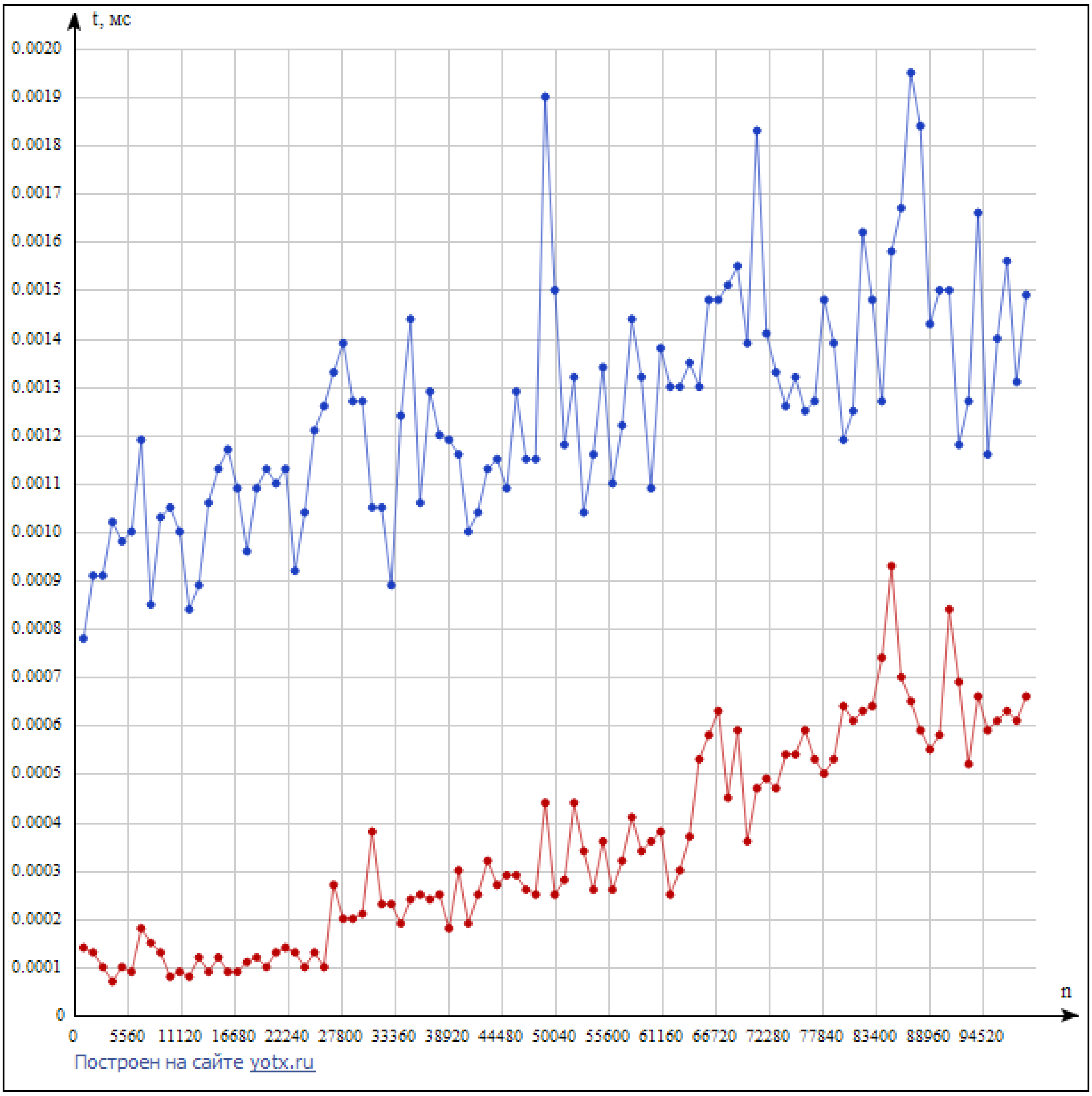


Рисунок 21 — Сравнение зависимостей времени добавления элемента от количества элементов для АВЛ-дерева (синяя) и хеш-таблицы с методом цепочек (красная).

По графику видно, что вначале для добавления элемента в хеш-таблицу требуется меньше времени чем для добавления в АВЛ-дерево, но с увеличением количества элементов время, требуемое для добавления в хеш-таблицу, растет быстрее времени, требуемого для добавления в АВЛ-дерево, что подтверждает теоретическую оценку.

Сравнение удалений элементов

Теоретически удаление элемента из АВЛ-дерево должно выполняться за O(log n), а из хеш-таблицы с методом цепочек за O(1+*a*), т. к. в обоих случаях перед удалением элемента необходимо произвести поиск этого элемента. Следовательно время удаления из хеш-таблицы должно быть меньше веремени удаления из АВЛ-дерева при малом количестве элементов. Для исследования были произведены замеры времени удаления случайного элемента для АВЛ-дерева и хеш-таблицы с методом цепочек при добавлении каждого 1000 элемента. Результаты представлены на рис. 22.

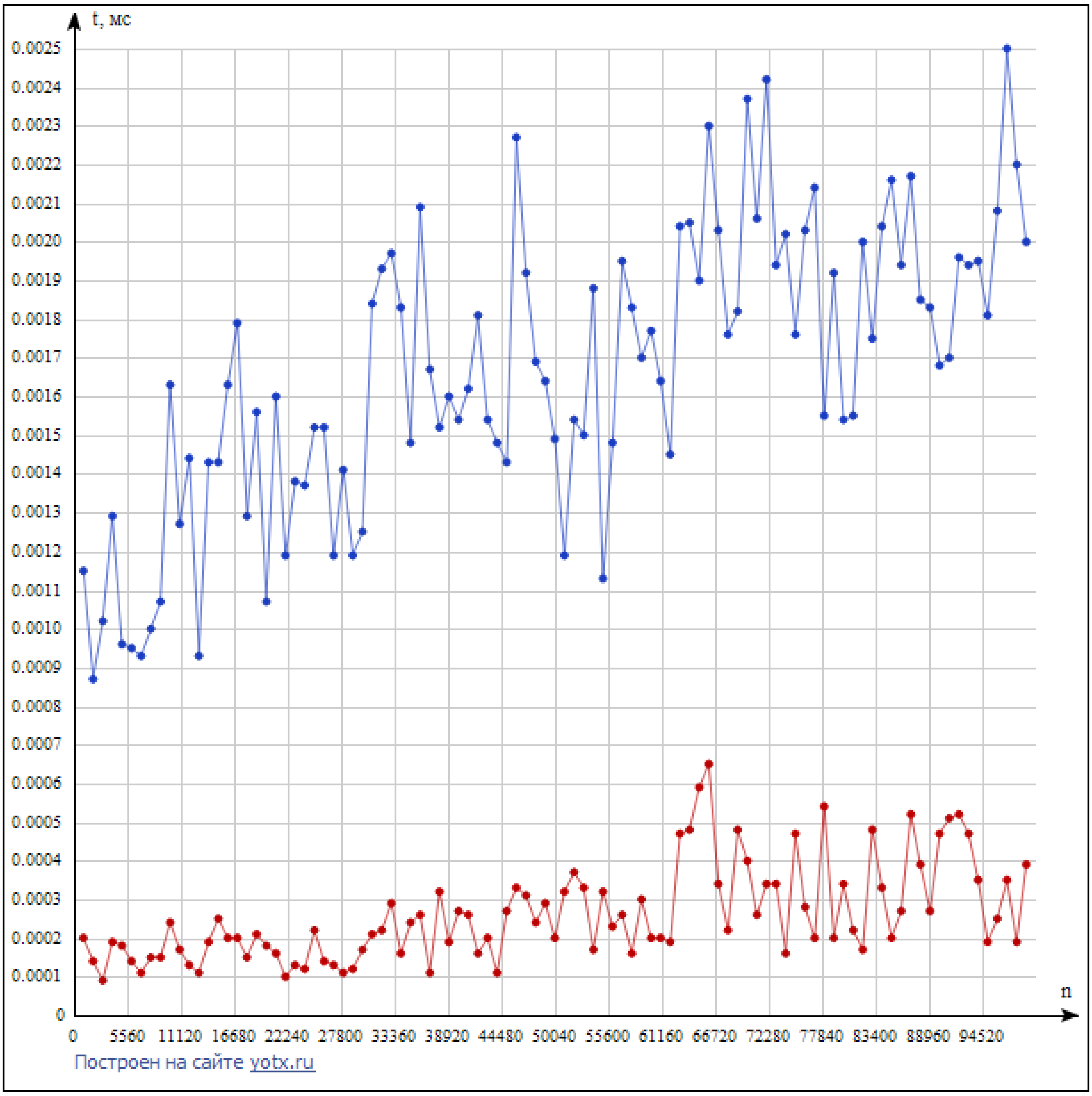


Рисунок 22 — Сравнение зависимости времени удаления от количетсва элементов для АВЛ-дерева (синяя) и хеш-таблицы с методом цепочек (крассная).

По рисунку видно, что для удаления случайного элемента из хеш-таблицы с методом цепочек требует меньше времени чем удаление из АВЛ-дерева, что подтверждает теоретическую оценку.

ЗАКЛЮЧЕНИЕ

В курсовой работе на языке Python были реализованы такие абстрактные структуры данных, как АВЛ-дерево и хеш-таблица с методом цепочек. Была проведена проверка корректности реализации обоих структур. Были проведены исследования каждой структуры, а именно исследовано время поиска, добавления и удаления элементов. Было проведено сравнение АВЛ-дерева и хеш-таблицы с методом цепочек по зависимости времени удаления, добавления и поиска элементов от количества элементов.

Исследвоание подтвердило теоретические оценки, за которые должны волнять операции, а именно для АВЛ-дерева операции удаления, добавления и поиска за O(log n), а для хеш-таблицы с методом цепочек эти операции в среднем случае за $O(1+a)$, в лучшем за O(1). По результатам сравнения было выявлено, что при малом количестве элементов операции удаления, добавления и поиска элементов в хеш-таблице выполняются быстрее аналогичных операций в АВЛ-дереве, но с увеличением количества элементов, время требуемое для операций с хеш-таблицой увеличивается быстрее такого же времени для АВЛ-дерева.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Статья об АВЛ-деревьях на Хабре // habr.com URL: https://habr.com/ru/post/150732/

2. Статья о хеш-таблице с методом цепочек на CodeChick // codechick.io URL: https://codechick.io/tutorials/dsa/dsa-hash-table

3. Онлайн-справочник по языку Python // PythonWorld URL: https://pythonworld.ru/

ПРИЛОЖЕНИЕ А

НАЗВАНИЕ ПРИЛОЖЕНИЯ

Файл: Hash/hash.py

```
import time

class Node:
    def __init__(self, key):
        self.key = key
        self.child = None
        self.parent = None

    def __str__(self):
        return "Key: {}, parent: {}, child: {}".format(self.key, self.parent, self.child)

class Hash_table:
    def __init__(self):
        self.P = 7757
        self.table = [Node(None) for a in range(self.P)]

    def hash_func(self, key):
        return key % self.P

    def add_key(self, key):
        hash = self.hash_func(key)
        new_node = Node(key)
        if self.table[hash].key == None:
            self.table[hash] = new_node
        else:
            node = self.table[hash]
            while True:
                if node.child == None:
                    node.child = new_node
                    new_node.parent = node
                    break
                elif node.key == key:
                    break
                else:
                    node = node.child

    def add_key_time(self, key):
        time_start = time.perf_counter()
        hash = self.hash_func(key)
        new_node = Node(key)
        if self.table[hash].key == None:
            self.table[hash] = new_node
        else:
            node = self.table[hash]
            while True:
                if node.child == None:
                    node.child = new_node
                    new_node.parent = node
                    break
                elif node.key == key:
```

```
                break
            else:
                node = node.child
        dif_time = (time.perf_counter() - time_start) * 100
        #print(f"Время, затраченное для добавление элемента - {dif_time:0.4f} мс")
        return dif_time

    def search_key(self, key):
        hash = self.hash_func(key)
        node = self.table[hash]
        while True:
            if node.key == key:
                return node
            else:
                node = node.child
                if node == None:
                    print("Элемента с ключом {} в хэш-таблице нет".format(key))
                    return Node(None)

    def search_key_time(self, key):
        time_start = time.perf_counter()
        hash = self.hash_func(key)
        node = self.table[hash]
        while True:
            if node.key == key:
                dif_time = (time.perf_counter() - time_start) * 100
                #print(f"Время, затраченное для поиск элемента - {dif_time:0.4f} мс")
                return dif_time
            else:
                node = node.child
                if node == None:
                    print("Элемента с ключом {} в хэш-таблице нет".format(key))
                    dif_time = (time.perf_counter() - time_start) * 100
                    #print(f"Время, затраченное для поиск элемента - {dif_time:0.4f} мс")
                    return dif_time

    def delete_key(self, key):
        hash = self.hash_func(key)
        node = self.search_key(key)
        if not node:
            return
        else:
            p_node = node.parent
            c_node = node.child
            if not p_node:
                if not c_node:
                    self.table[hash] = Node(None)
                else:
                    self.table[hash] = c_node
            else:
                p_node.child = c_node

    def delete_key_time(self, key):
        time_start = time.perf_counter()
        hash = self.hash_func(key)
        node = self.search_key(key)
```

```
        if not node:
            dif_time = (time.perf_counter() - time_start) * 100
            #print(f"Время, затраченное для удаления элемента - {dif_time:0.4f} мс")
            return dif_time
        else:
            p_node = node.parent
            c_node = node.child
            if not p_node:
                if not c_node:
                    self.table[hash] = Node(None)
                else:
                    self.table[hash] = c_node
            else:
                p_node.child = c_node
        dif_time = (time.perf_counter() - time_start) * 100
        #print(f"Время, затраченное для удаления элемента - {dif_time:0.4f} мс")
        return dif_time

if __name__ == '__main__':
    #nodes = list(map(int, input().split()))
    nodes = [a for a in range(1000000)]
    hash_table = Hash_table()
    for a in nodes:
        hash_table.add_key(a)
    hash_table.search_key_time(845554)
```

Файл: AVL/avl_tree.py

```
import time

class Node:
    def __init__(self, key):
        self.key = key
        self.left_child = None
        self.right_child = None
        self.height = 0

    def __str__(self):
        left = self.left_child.key if self.left_child else None
        right = self.right_child.key if self.right_child else None
        return 'key: {}, left: {}, right: {}'.format(self.key, left, right)

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

    def print_tree(self):
        self.print_AVL(False, '', self.root)

    def print_AVL(self, isL, prefix, node):
        if(node != None):
            print(prefix, end='')
            if isL:
                print('└── L ', end='')
            else:
```

```
                print('└── R ', end='')
            print(node.key)
            if isL:
                prefix += '│   '
            else:
                prefix += '    '
            self.print_AVL(True, prefix, node.left_child)
            self.print_AVL(False, prefix, node.right_child)

    def add_key(self, key):
        if not self.root:
            self.root = Node(key)
        else:
            self.root = self.add_node(key, self.root)

    def add_key_time(self, key):
        time_start = time.perf_counter()
        if not self.root:
            self.root = Node(key)
        else:
            self.root = self.add_node(key, self.root)
        dif_time = (time.perf_counter() - time_start) * 100
        return dif_time
        #print(f"Время, затраченное для добавление элемента - {dif_time:0.4f} мс")

    def add_node(self, key, node):
        if not node:
            node = Node(key)
        elif key < node.key:
            node.left_child = self.add_node(key, node.left_child)
            if self.get_height(node.left_child) - self.get_height(node.right_child) >= 2:
                if key < node.left_child.key:
                    node = self.small_right_rotate(node)
                else:
                    node = self.big_right_rotate(node)
        elif key > node.key:
            node.right_child = self.add_node(key, node.right_child)
            if self.get_height(node.right_child) - self.get_height(node.left_child) >= 2:
                if key < node.right_child.key:
                    node = self.big_left_rotate(node)
                else:
                    node = self.small_left_rotate(node)
        else:
            return node
        node.height = max(self.get_height(node.left_child),
self.get_height(node.right_child)) + 1
        return node

    def get_height(self, node):
        if not node:
            return 0
        else:
            return node.height

    def small_left_rotate(self, node_a):
        node_b = node_a.right_child
```

```
        node_a.right_child = node_b.left_child
        node_b.left_child = node_a

        node_a.height = max(self.get_height(node_a.right_child),
self.get_height(node_a.left_child)) + 1
        node_b.height = max(self.get_height(node_b.right_child),
self.get_height(node_b.left_child)) + 1

        return node_b

    def small_right_rotate(self, node_b):
        node_a = node_b.left_child
        node_b.left_child = node_a.right_child
        node_a.right_child = node_b

        node_a.height = max(self.get_height(node_a.right_child),
self.get_height(node_a.left_child)) + 1
        node_b.height = max(self.get_height(node_b.right_child),
self.get_height(node_b.left_child)) + 1

        return node_a

    def big_left_rotate(self, node_a):
        node_b = node_a.right_child
        node_a.right_child = self.small_right_rotate(node_b)
        node_c = self.small_left_rotate(node_a)

        return node_c

    def big_right_rotate(self, node_a):
        node_b = node_a.left_child
        node_a.left_child = self.small_left_rotate(node_b)
        node_c = self.small_right_rotate(node_a)

        return node_c

    def get_root(self):
        return self.root

    def search_key(self, key):
        node = self.root
        while True:
            if key == node.key:
                return node
            else:
                if key < node.key:
                    if node.left_child == None:
                        print("Узла с ключом {} нет в дереве".format(key))
                        return Node(-1)
                    node = node.left_child
                else:
                    if node.right_child == None:
                        print("Узла с ключом {} нет в дереве".format(key))
                        return Node(-1)
                    node = node.right_child
```

```
    def search_key_time(self, key):
        time_start = time.perf_counter()
        node = self.root
        while True:
            if key == node.key:
                dif_time = (time.perf_counter() - time_start) * 100
                #print(f"Время, затраченное для поиск элемента - {dif_time:0.4f} мс")
                return dif_time
            else:
                if key < node.key:
                    if node.left_child == None:
                        #print("Узла с ключом {} нет в дереве".format(key))
                        dif_time = (time.perf_counter() - time_start) * 100
                        #print(f"Время, затраченное для поиск элемента - {dif_time:0.4f}
мс")
                        return dif_time
                    node = node.left_child
                else:
                    if node.right_child == None:
                        #print("Узла с ключом {} нет в дереве".format(key))
                        dif_time = (time.perf_counter() - time_start) * 100
                        #print(f"Время, затраченное для поиск элемента - {dif_time:0.4f}
мс")
                        return dif_time
                    node = node.right_child

    def delete_key(self, key):
        if self.search_key(key).key == -1:
            return
        else:
            self.root = self.delete_node(key, self.root)

    def delete_key_time(self, key):
        time_start = time.perf_counter()
        if self.search_key(key).key == -1:
            return
        else:
            self.root = self.delete_node(key, self.root)
        dif_time = (time.perf_counter() - time_start) * 100
        #print(f"Время, затраченное для удаления элемента - {dif_time:0.4f} мс")
        return dif_time

    def delete_node(self, key, node):
        if not node:
            return None
        if key < node.key:
            node.left_child = self.delete_node(key, node.left_child)
        elif key > node.key:
            node.right_child = self.delete_node(key, node.right_child)
        else:
            node_L = node.left_child
            node_R = node.right_child
            if node_R == None:
                return node_L
            min_node = self.search_min(node_R)
            min_node.right_child = self.remove_min(node_R)
```

```
            min_node.left_child = node_L
            if self.root.key == key:
                self.root = min_node
            return self.balance(min_node)
        return self.balance(node)

    def remove_min(self, node):
        if not node.left_child:
            return node.right_child
        node.left_child = self.remove_min(node.left_child)
        return self.balance(node)

    def search_min(self, node):
        if node.left_child:
            return self.search_min(node.left_child)
        else:
            return node

    def balance(self, node):
        #node.height = max(self.get_height(node.left_child),
self.get_height(node.right_child)) + 1
        if self.get_height(node.right_child) - self.get_height(node.left_child) >= 2:
            if self.get_height(node.right_child.right_child) -
self.get_height(node.right_child.left_child) < 0:
                node.right_child = self.small_right_rotate(node.right_child)
            return self.small_left_rotate(node)

        if self.get_height(node.left_child) - self.get_height(node.right_child) >= 2:
            if self.get_height(node.left_child.right_child) -
self.get_height(node.left_child.left_child) > 0:
                node.left_child = self.small_left_rotate(node.left_child)
            return self.small_right_rotate(node)

        return node


if __name__ == '__main__':
    nodes = list(map(int, input().split()))
    avl_tree = AVLTree()
    for index, node in enumerate(nodes):
        avl_tree.add_key(node)
    avl_tree.print_tree()
    avl_tree.delete_key(5)
    avl_tree.print_tree()
```

Файл: tests_hash.py

```
from Hash.hash import Node, Hash_table

def test_delete():
    hash_table = Hash_table()
    for a in range(10000):
        hash_table.add_key(a)
    hash_table.delete_key(8888)
```

```
def test_search():
    hash_table = Hash_table()
    for a in range(10000):
        hash_table.add_key(a)
    hash_table.search_key(9999)

def test_add():
    hash_table = Hash_table()
    for a in range(10000):
        hash_table.add_key(a)
    hash_table.add_key(10010010)
```

Файл: tests_avl.py

```
from AVL.avl_tree import Node, AVLTree

def test_delete():
    avl_tree = AVLTree()
    for a in range(10000):
        avl_tree.add_key(a)
    avl_tree.delete_key(5874)

def test_search():
    avl_tree = AVLTree()
    for a in range(10000):
        avl_tree.add_key(a)
    avl_tree.search_key(741)

def test_add():
    avl_tree = AVLTree()
    for a in range(10000):
        avl_tree.add_key(a)
    avl_tree.add_key(897561)
```

Файл: research.py

```
import random

from AVL.avl_tree import Node, AVLTree
from Hash.hash import Node, Hash_table

def search_res():
    hash_file = open("hash_sear.txt", "w")
    avl_file = open("avl_sear.txt", "w")
    hash_table = Hash_table()
    avl_tree = AVLTree()

    for a in range(100000):
        hash_table.add_key(a)
        avl_tree.add_key(a)
        if not a % 1000 and a != 0:
            key = random.randint(0, a)
            hash_time = hash_table.search_key_time(key)
            avl_time = avl_tree.search_key_time(key)
```

```
            hash_file.write(str(a) + " " + str(hash_time) + "\n")
            avl_file.write(str(a) + " " + str(avl_time) + "\n")

def add_res():
    hash_file = open("hash_add.txt", "w")
    avl_file = open("avl_add.txt", "w")
    hash_table = Hash_table()
    avl_tree = AVLTree()

    for a in range(100000):
        hash_table.add_key(a)
        avl_tree.add_key(a)
        if not a % 1000 and a != 0:
            key = random.randint(100001, 101001)
            hash_time = hash_table.add_key_time(key)
            avl_time = avl_tree.add_key_time(key)
            hash_file.write(str(a) + " " + str(hash_time) + "\n")
            avl_file.write(str(a) + " " + str(avl_time) + "\n")


def delete_res():
    hash_file = open("hash_del.txt", "w")
    avl_file = open("avl_del.txt", "w")
    hash_table = Hash_table()
    avl_tree = AVLTree()

    for a in range(100000):
        hash_table.add_key(a)
        avl_tree.add_key(a)
        if not a % 1000 and a != 0:
            key = random.randint(0, a)
            hash_time = hash_table.delete_key_time(key)
            avl_time = avl_tree.delete_key_time(key)
            hash_file.write(str(a) + " " + str(hash_time) + "\n")
            avl_file.write(str(a) + " " + str(avl_time) + "\n")


if __name__ == "__main__":
    search_res()
    delete_res()
    add_res()
```