Словари. Отчёт.

Задача

Реализовать несколько способов организации словаря (массивы с линейным поиском, упорядоченные массивы с бинарным поиском, деревья, сбалансированные деревья, хеш-таблицы) и сравнить их (время выполнения операций со словарем, объем занимаемой памяти) между собой и со встроенным dict.

Параметры вычислительного узла

- *Intel Core i7-2600 CPU @ 3.40GHz*
- 16,0GB RAM
- MS Windows 7 64-bit SP1

Описание тестируемых алгоритмов

Словарь с линейным поиском:

- *В основе словаря лежит массив.*
- *При добавлении ключа необходимо "пройти" по всему массиву (если добавляемый ключ уже был в словаре), а затем добавить новый ключ (или обновить значение старого).*
- *При поиске ключа необходимо "пройти" по всему массиву.*

Словарь с бинарным поиском:

- *В основе словаря лежит упорядоченный массив.*
- *При добавлении ключа пользуемся бинарным поиском и добавляем новый ключ/обновляем значение старого.*
- *При поиске ключа пользуемся бинарным поиском.*

Словарь на БИНАРНОМ ДЕРЕВЕ:

- *В основе словаря лежит бинарное дерево поиска.*
- *Выполняются условия:*
  - *Оба поддерева - левое и правое - являются бинарными деревьями поиска.*
  - *У всех узлов левого поддерева произвольного узла X значения ключей данных **меньше**, нежели значение ключа данных самого узла X.*
  - *У всех узлов правого поддерева произвольного узла X значения ключей данных **не меньше**, нежели значение ключа данных самого узла X.*

- *При добавлении ключа спускаемся по дереву в нужном направлении и добавляем новый ключ/обновляем значение старого.*
- *При поиске ключа спускаемся по дереву в нужном направлении, пока не найдём ключ.*

Словарь на AVL-дереве:

- *В основе словаря лежит сбалансированное бинарное дерево поиска.*
- *Выполняется условие:*
  - *Для каждой вершины: высота её двух поддеревьев отличается не более, чем на единицу.*
- *При добавлении ключа спускаемся по дереву в нужном направлении и добавляем новый ключ/обновляем значение старого. После этого рекурсивно балансируем дерево.*
- *При поиске ключа спускаемся по дереву в нужном направлении, пока не найдём ключ.*

Хеш-таблица (открытая адресация):

- *В основе словаря лежит два массива. Один отвечает за хранение пар ключ-значение, второй же хранит информацию об удалённых элементах (булев массив)*
- *Выполнение операции вставки и поиска ключа начинается с вычисления хеш-функции от ключа. Получившееся значение играет роль индекса в массиве.*
- *Добавляем/обновляем значение ключа по найденному индексу.*
- *При поиске извлекаем значение по найденному индексу.*

BUILT-IN DICTIONARY

Результаты тестирования

- Первый тест. Все реализации словарей. Объём данных - 7 000, шаг - 100, повторения - 15.
  - Линейный поиск (**коричневый** цвет)
  - Бинарный поиск (**оранжевый** цвет)
  - Бинарное дерево (**зеленый** цвет)
  - AVL-дерево (**синий** цвет)
  - Хеш-таблица (**фиолетовый** цвет)
  - built-in dictionary (**красный** цвет)

- Добавление ключей (кликабельно)

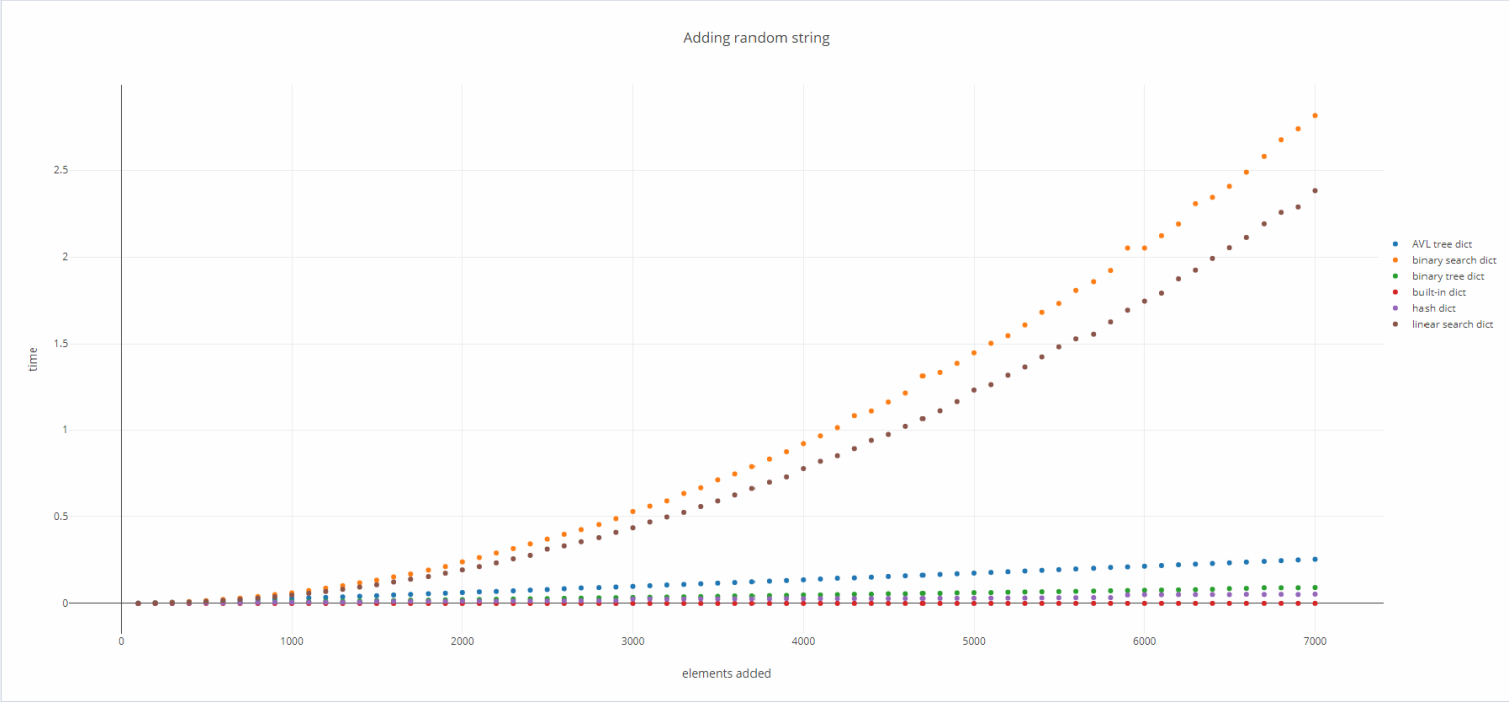

График 1.1.1 Вставка ключей

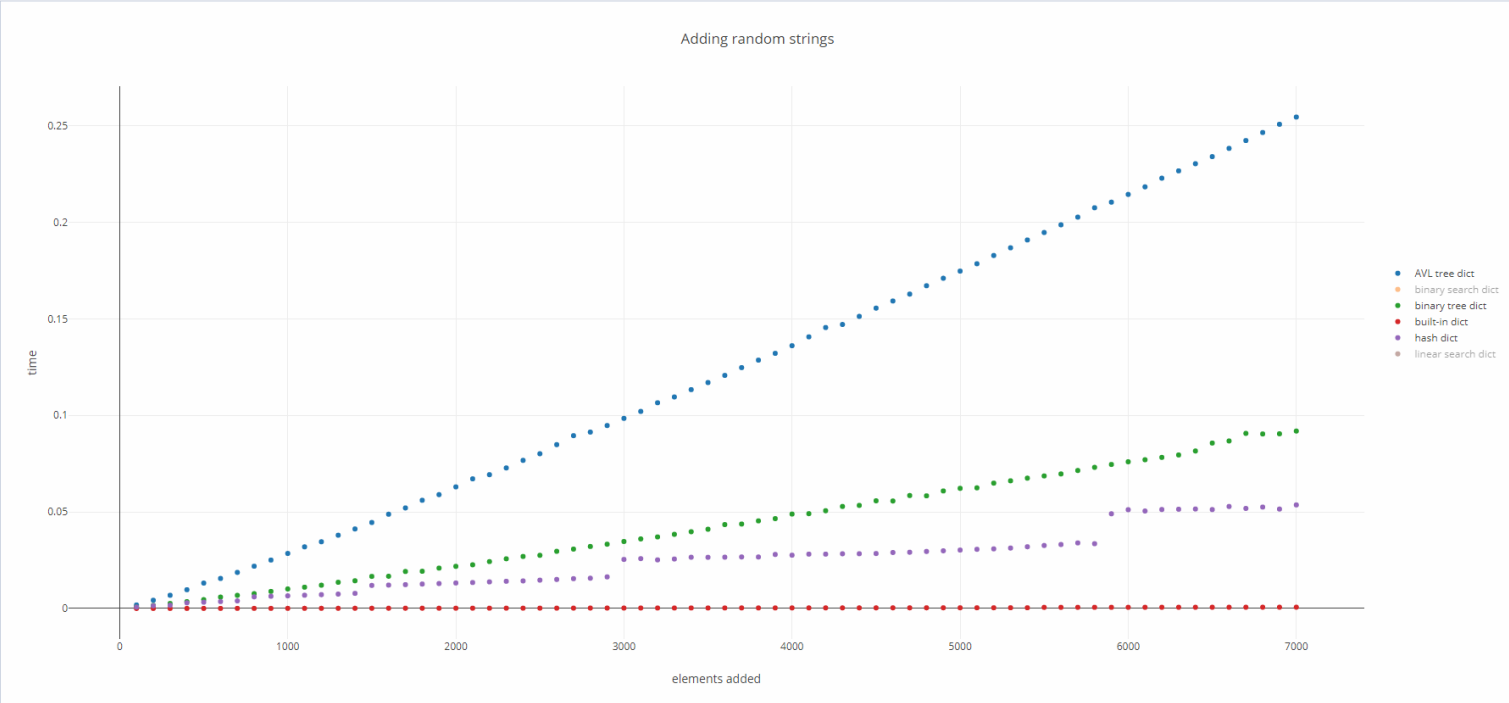

График 1.1.2 Вставка ключей (скрыты линейный и бинарный поиски)

- Поиск ключей (кликабельно)

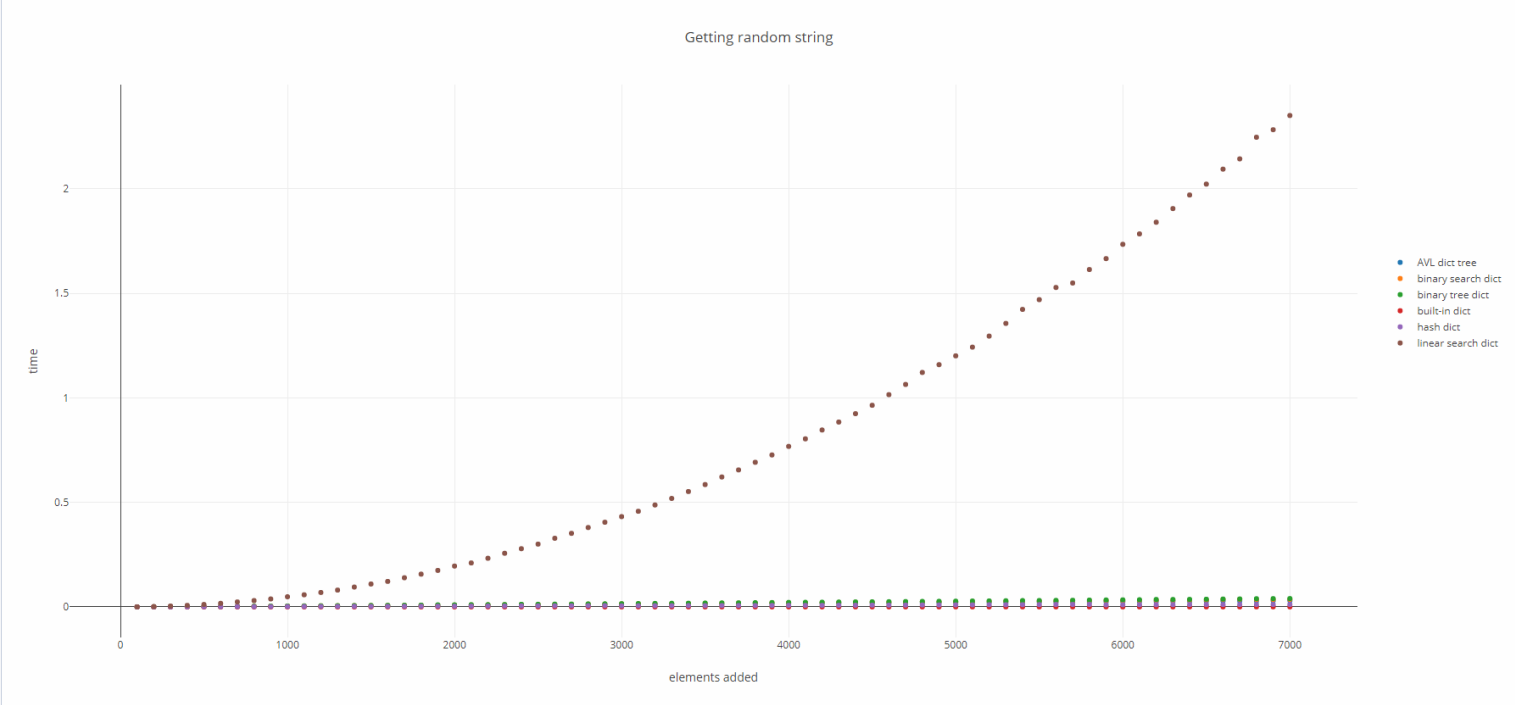

График 1.2.1 Поиск ключей

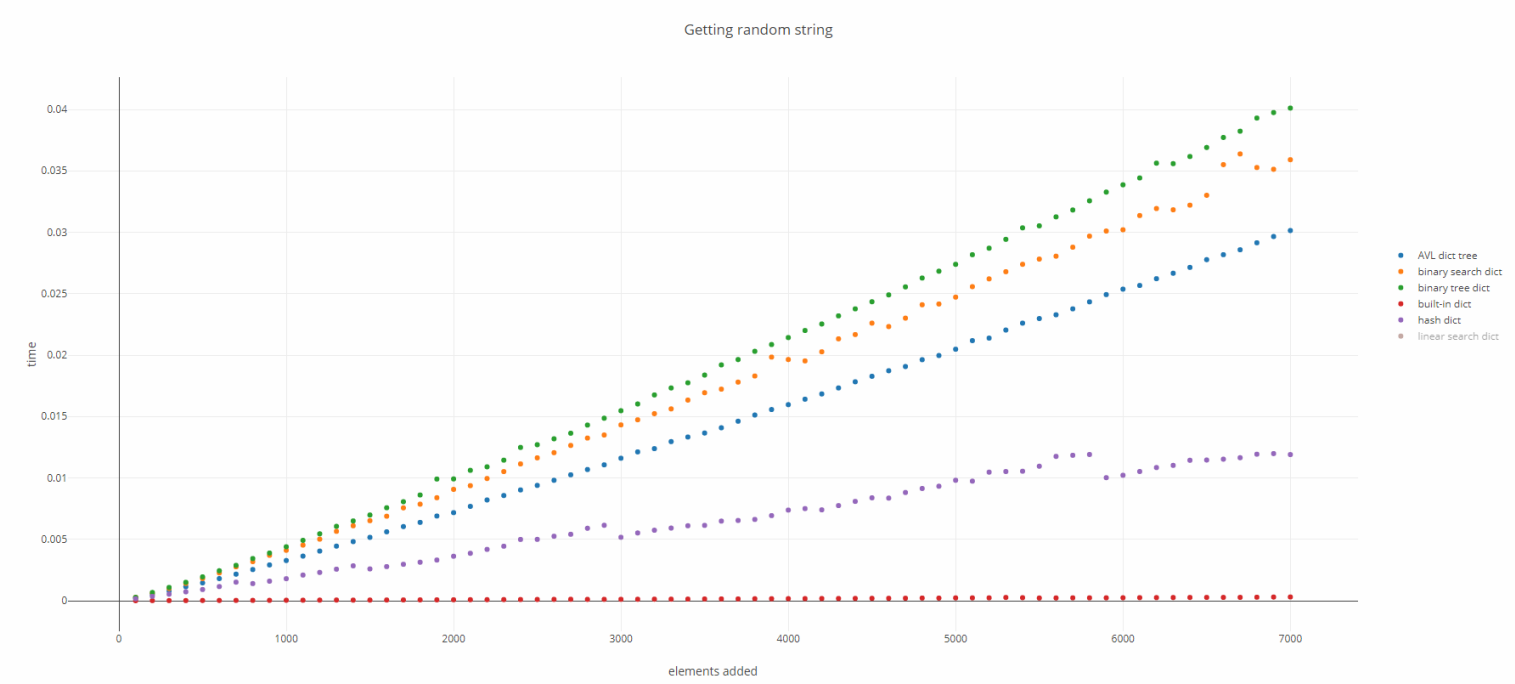

График 1.2.2. Поиск ключей (скрыт линейный поиск)

- Второй тест. Словари на деревьях, хеш-таблицы. Объём данных - 100 000, шаг - 100, повторения - 10.
  - Бинарное дерево (**оранжевый** цвет)
  - AVL-дерево (**синий** цвет)
  - Хеш-таблица (**красный** цвет)
  - built-in dictionary (**зеленый** цвет)

  - Добавление ключей (кликабельно)

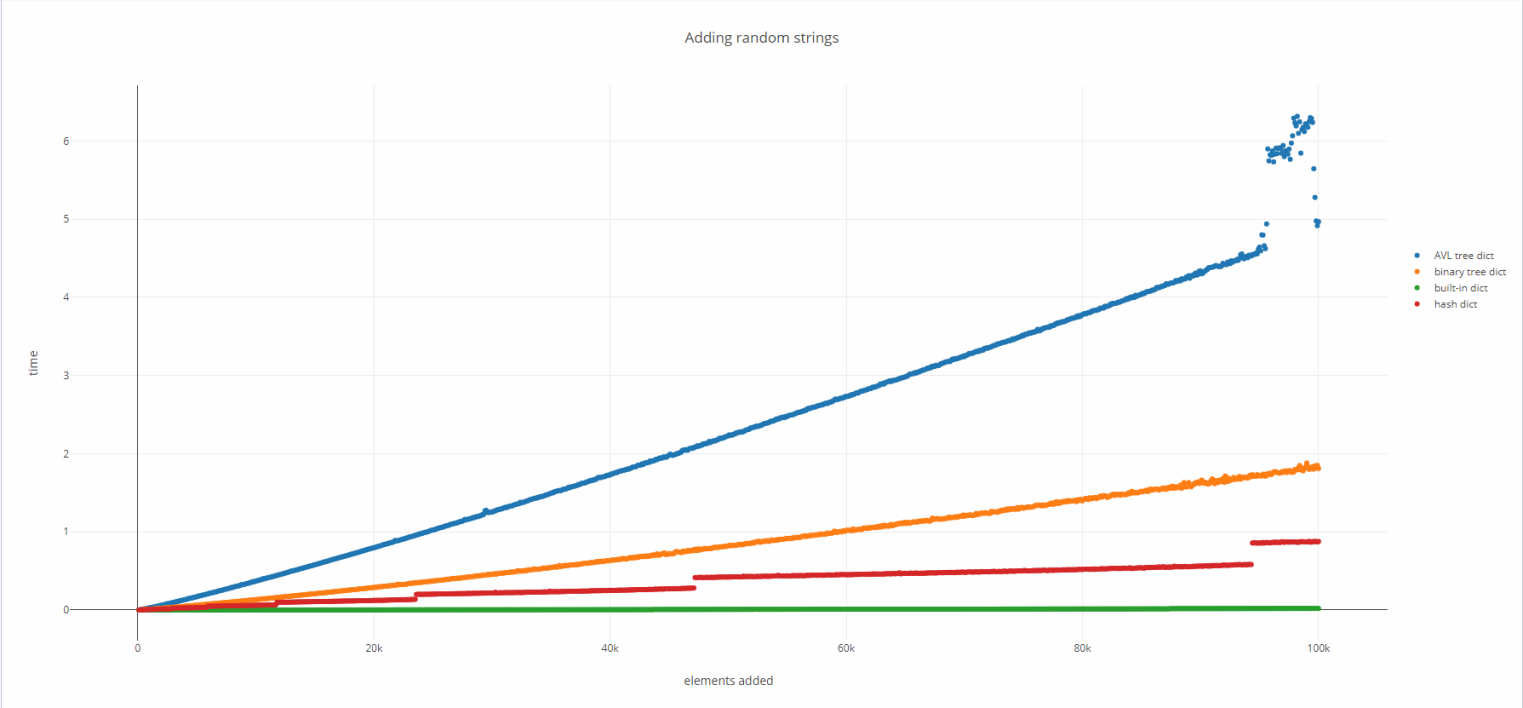

График 2.1. Вставка ключей

- Поиск ключей (кликабельно)

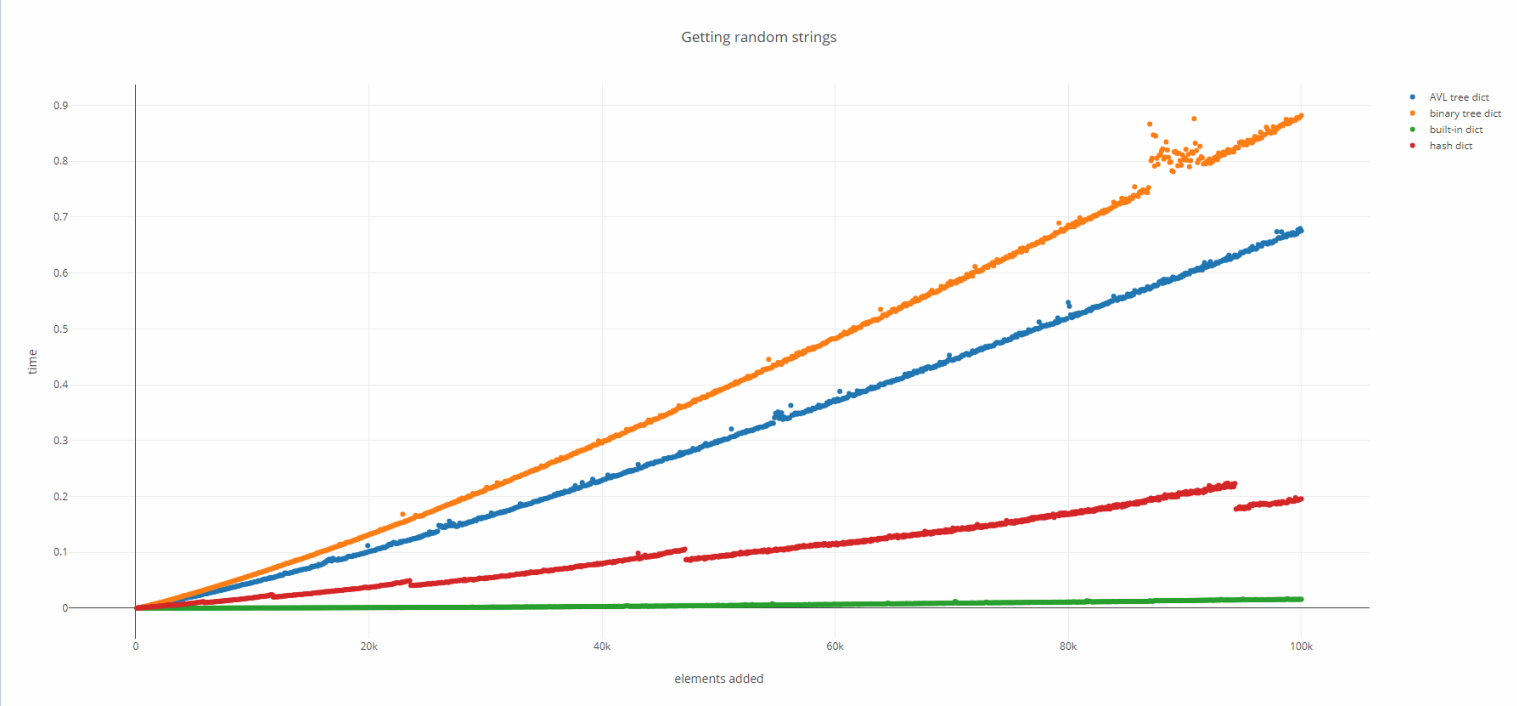

График 2.2. Поиск ключей

- Третий тест. Хеш-таблицы. Объём данных - 5 000, шаг - 200, повторения - 2.
  В этом тесте использовались разные хеш-функции:
  - Константная хеш-функция (**оранжевый** цвет)
  - Хеш-функция, возвращающая длину введённого значения (**синий** цвет)
  - Встроенная хеш-функция (**зелёный** цвет)

- Добавление ключей

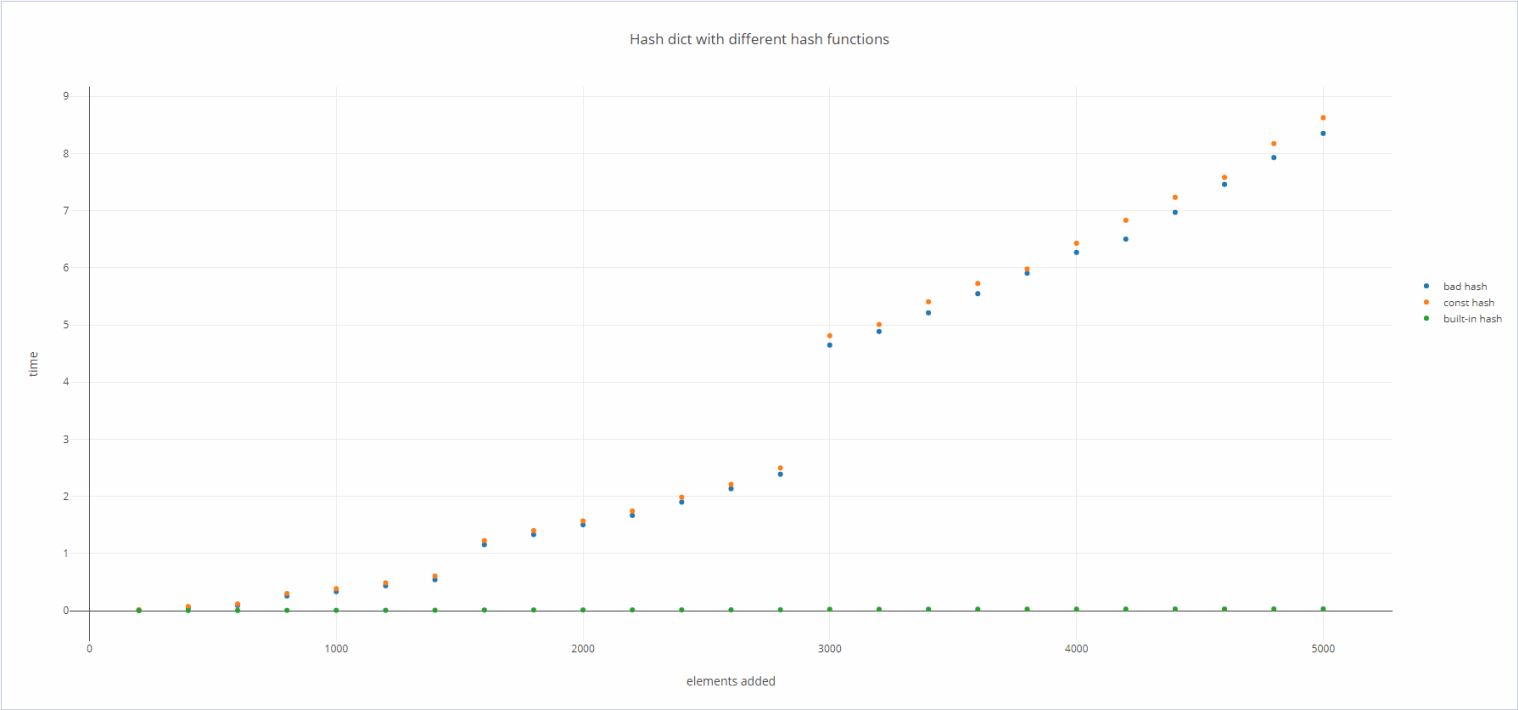

График 3.1. Вставка ключей

- Поиск ключей

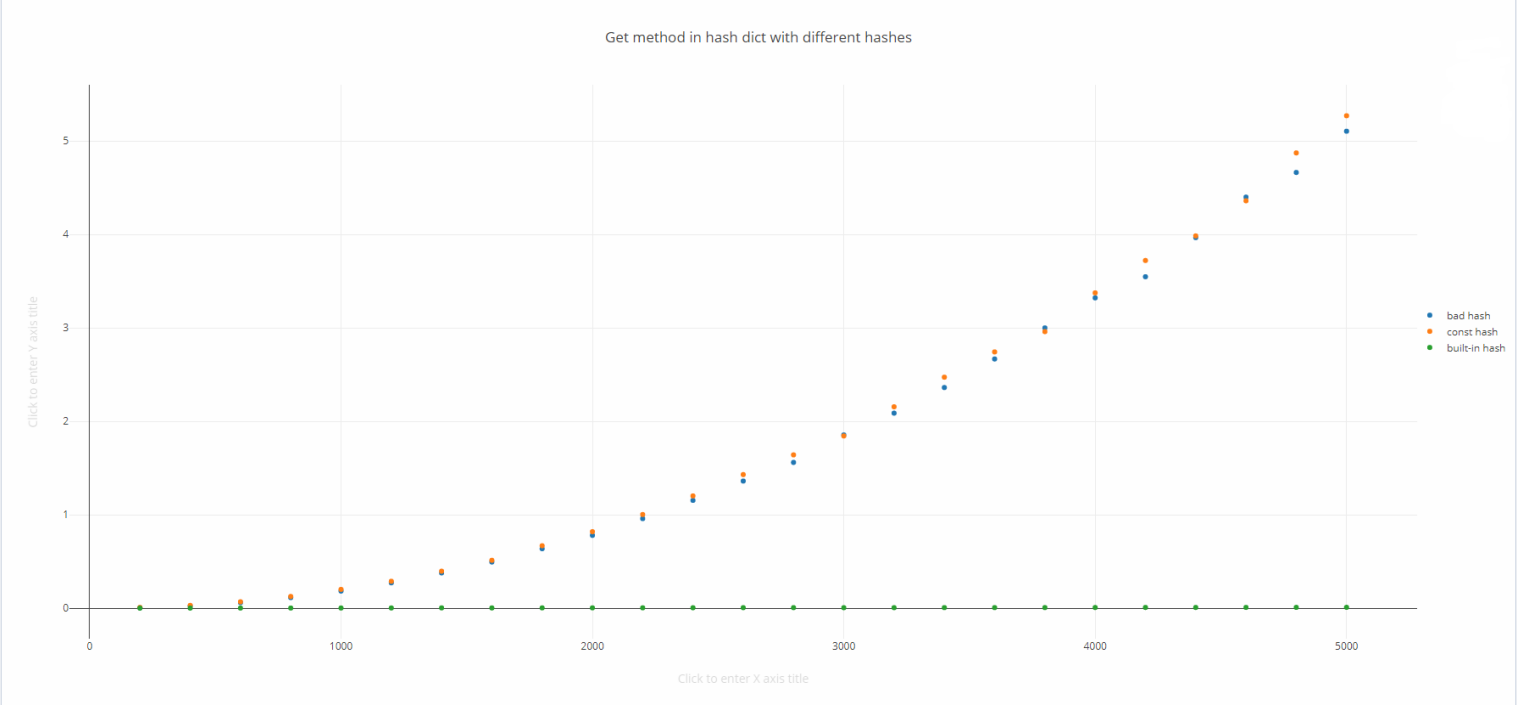

График 3.2. Поиск ключей

❖ Четвёртый тест. Хеш-таблицы. Объём данных - 50 000, шаг - 200, повторения - 12. В этом тесте у хеш-таблиц разный параметр load factor (степень заполненности, при котором хеш-таблица расширится и перезапишет все значения)

- Load factor 0.26 (**синий** цвет)
- Load factor 0.4 (**оранжевый** цвет)
- Load factor 0.52 (**зеленый** цвет)
- Load factor 0.72 (**красный** цвет)
- Load factor 0.88 (**фиолетовый** цвет)
- Load factor 0.99 (**коричневый** цвет)

- Добавление ключей (кликабельно)

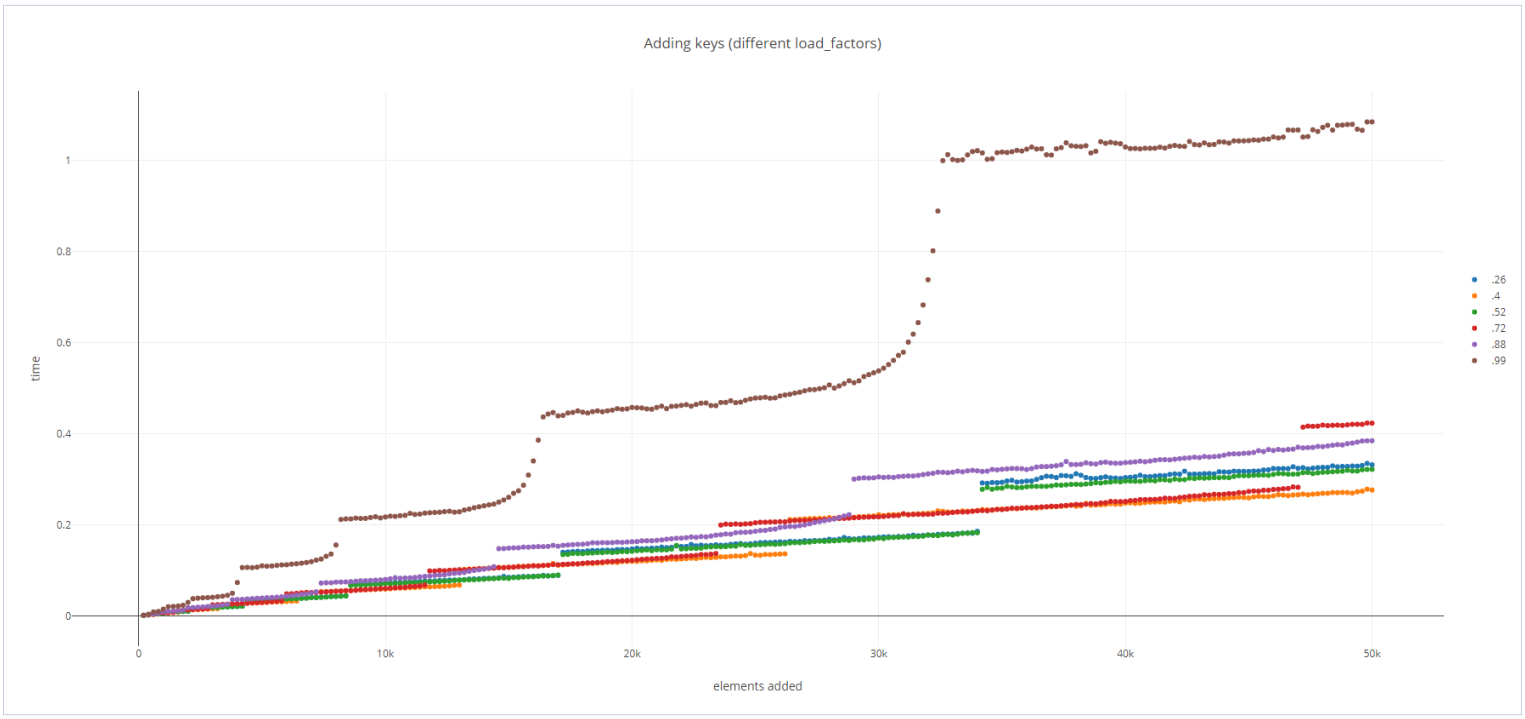

График 4.1.1. Вставка ключей

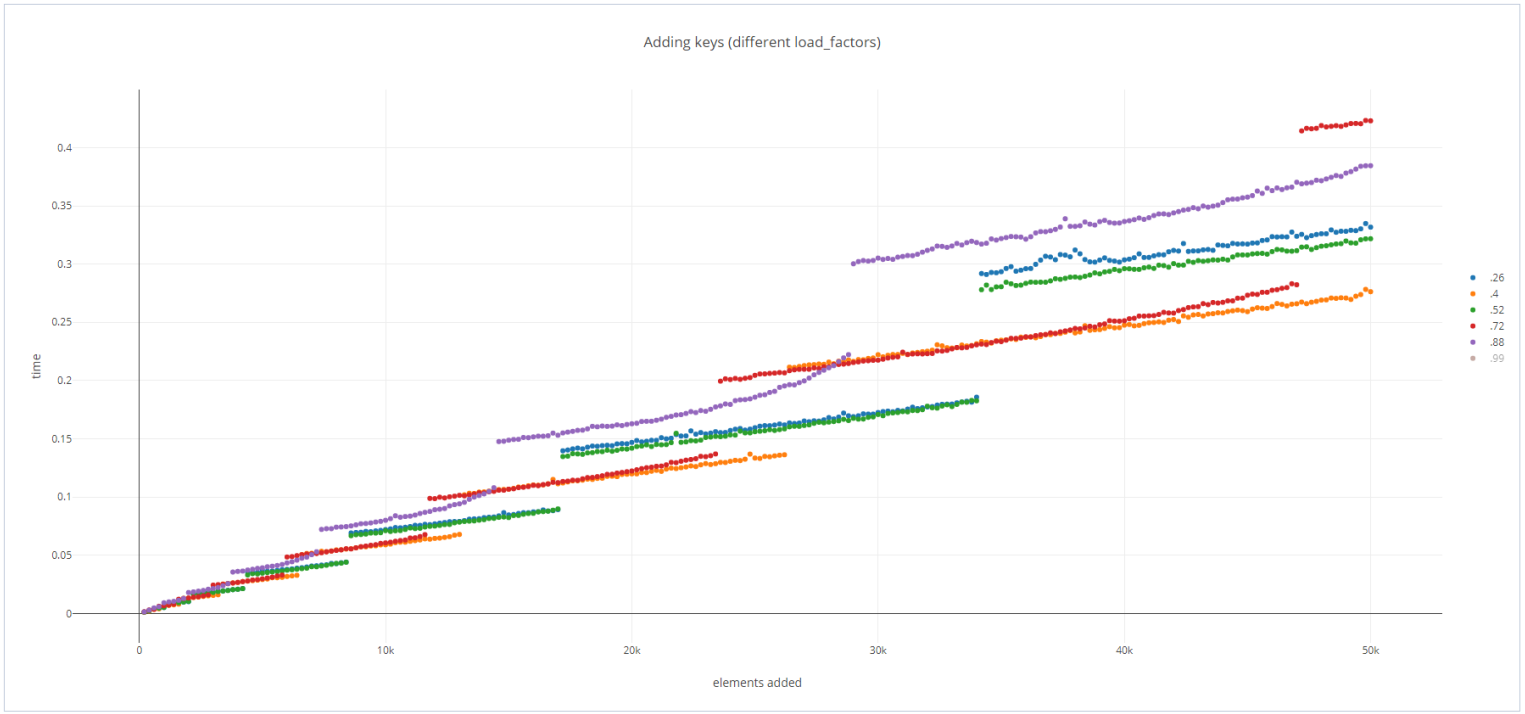

График 4.1.2. Вставка ключей (скрыт .99)

- Поиск ключей (кликабельно)

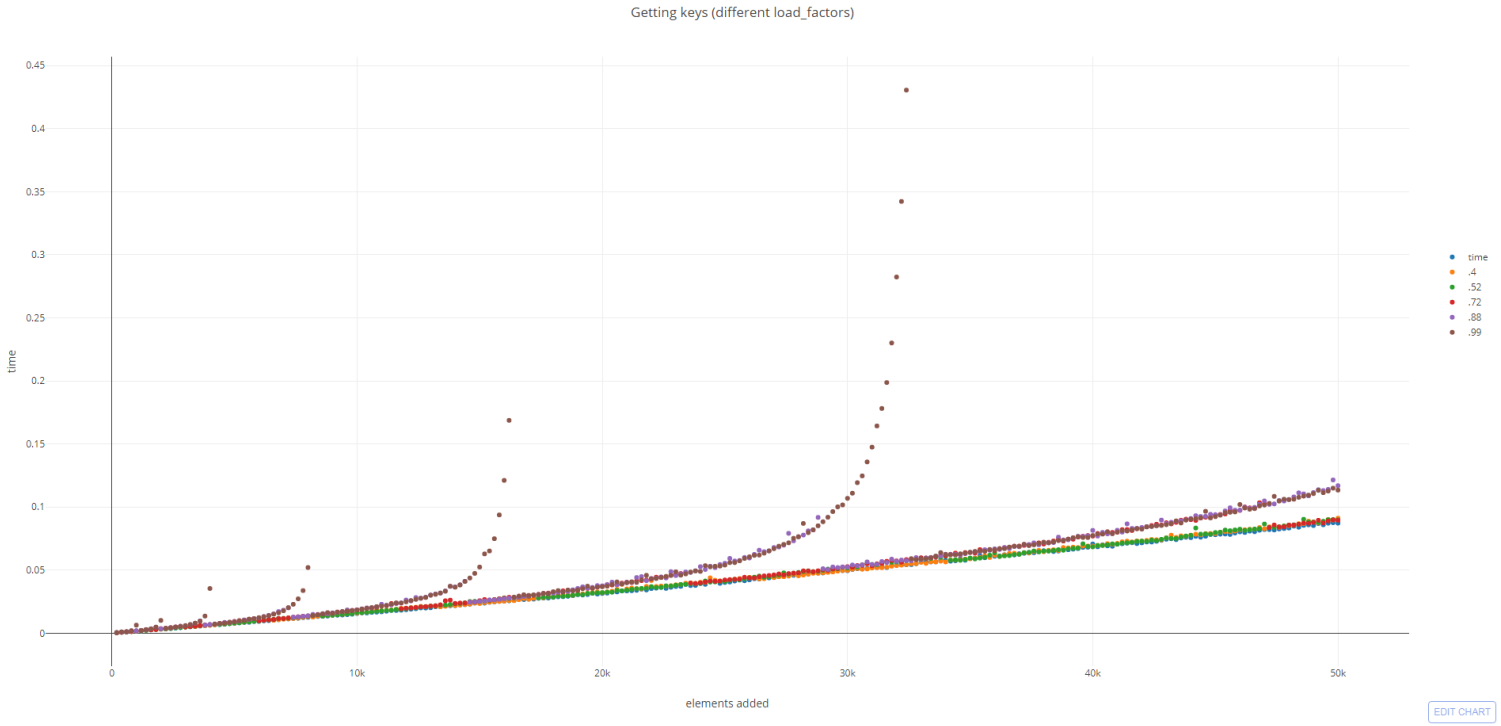

График 4.2.1. Поиск ключей

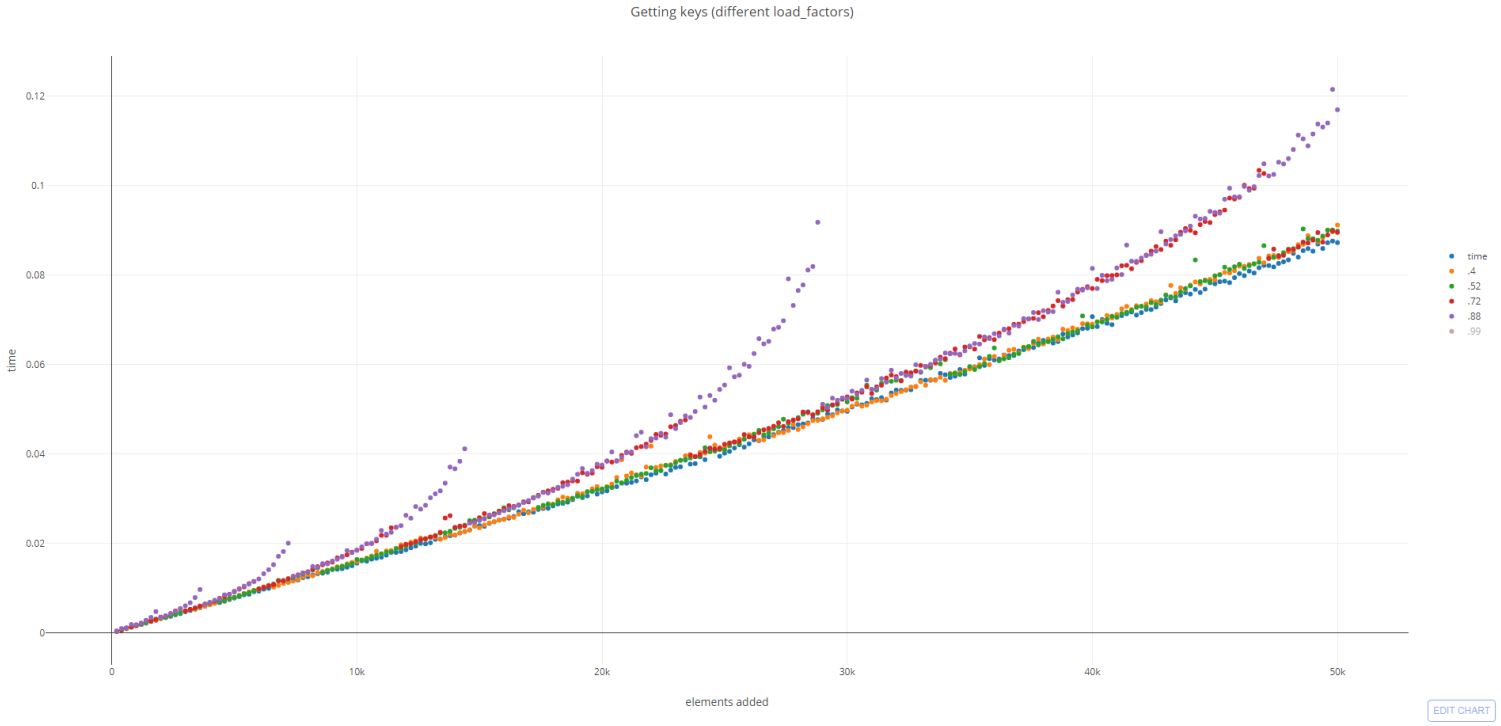

График 4.2.2. Поиск ключей (скрыт .99)

Обоснование некоторых результатов

Во втором тесте возникли некоторые необъяснимые результаты (График 2.1. Вставка ключей **AVL-дерево** и График 2.1. Вставка ключей **Бинарное дерево**). Несмотря на то, что при проведении тестирования процесс был изолирован, была завершена работа фоновых программ, некоторые внешние факторы (*ос, сторонние процессы*) всё-таки повлияли на результаты тестирования, однако лишь незначительным образом.

Быстродействие работы встроенного "*питонячего*" словаря находятся вне конкуренции. Этот словарь написан на *C*, поэтому прочие реализации словаря на *Python* (представленные в этой работе), безусловно, не сравнятся с ним.

При взгляде на результаты работы хеш-таблиц в глаза бросаются заметные *"скачки"*. Это обосновано тем, что при заполнении основного массива периодически приходится увеличивать его объём. Данная процедура влечёт за собой выделение нового участка памяти под массив и перезаполнение всех имеющихся ключей (на этом этапе удаляются ненужные ключи из словаря), что, конечно же, негативно сказывается на быстродействии добавлении нового ключа. Однако "разряжение" заполненности словаря имеет свои плюсы: поиск ключа в словаре происходит быстрее, за счёт меньшей вероятности возникновения коллизии. Это отчётливо видно на графиках результата работы хеш-таблицы при поиске ключа.

Анализ результатов

Несколько слов о формате тестирования:

> *Параметры тестирования имеют следующий вид: SUMMARY, STEP, REPEATS.*
>
> *SUMMARY - он же - верхний порог количества входных данных.*
>
> *STEP - шаг - величина, на которую увеличивается количество входных данных на каждой итерации (до SUMMARY включительно).*
>
> *REPEATS - повторения - столько раз будет выполнена каждая итерация (обеспечивает более точный результат).*

*Здесь и далее "**добавление**" означает операцию добавления ключа в словарь, "**поиск**" - операцию извлечения значения из словаря по заданному ключу.*

Первый тест. *Неудачный выбор.* *(SUMMARY=7 000, STEP=100, REPEATS=15)*

> Выделяются два результата: *линейный* и *бинарный поиски*. При **добавлении** обе структуры показали наихудшие результаты. Однако при **поиске** словарь на *отсортированном массиве* может конкурировать даже с деревьями (обоснование этому - алгоритм бинарного поиска и его сложность).

По результатам этого теста было решено в дальнейшем не рассматривать словари на *линейном* и *бинарном поиске*, так как с практической точки зрения данные реализации оказались крайне неподходящими для исследуемой задачи.

Второй тест. *Оптимизация.* *(SUMMARY=100 000, STEP=100, REPEATS=10)*

Рассмотрим *AVL-дерево* и *бинарное дерево поиска.* **Добавление** в *AVL-дереве* происходит медленнее, за счёт реализации самого алгоритма: каждый узел необходимо балансировать - возникает рекурсия (это негативно влияет на быстродействие). Здесь с большим отрывом выигрывает *бинарное дерево поиска*.

Взглянем на результаты **поиска** ключа. Здесь ситуация в точности противоположная - теряя время на балансировку узлов*, AVL-дерево* преобразует свою внутреннюю структуру таким образом, что поиск ключа становится более простой операцией.

Таким образом, на практике, для словаря больше подходит *AVL-дерево*, нежели *бинарное дерево поиска*, так как словарь можно построить один раз, а затем только искать в нём интересующие значения. Если же словарь используется как динамическая структура - то выбор падает на *бинарное дерево поиска*.

Теперь рассмотрим *деревья* и *хеш-таблицы.* Видно что *хеш-таблицы* справляются с задачей куда лучше, нежели *деревья*. В их основе лежит более простая структура и они больше подходят под конкретную задачу - **поиск** ключей.

Встроенный словарь, безусловно, является намного более совершенной версией написанной мною *хеш-таблицы*. Однако самописная *хеш-таблица* позволяет провести больше тестов, изменяя её внутренние параметры.

Итог второго теста предсказуем - для словаря следует использовать *хеш-таблицы*, и, в лучшем случае, *встроенные*.

Третий тест. *Важность хеш-функции.* *(SUMMARY=5 000, STEP=200, REPEATS=2)*

Хеш-функция - очень важная вещь в работе *хеш-таблицы*. Результаты её работы во многом определяют быстродействие как **добавления**, так и **поиска** ключа.

Взглянем на графики: используя плохую *(плохая - значит часто выдаёт коллизии) хеш-функцию* быстродействие словаря заметно снижается. Он буквально превращается в массив с линейным поиском, из-за того, что как при **добавлении**, так и при **поиске** ключа необходимо "пройти" почти по всему внутреннему массиву.

При разработке собственной *хеш-таблицы* стоит уделить особое внимание используемой *хеш-функции*, ведь от неё зависит, насколько эффективно будут выполняться базовые методы словаря.

Четвёртый тест. *Load factor.* *(SUMMARY=50 000, STEP=200, REPEATS=12)*

У *хеш-таблиц* есть ещё один важный параметр - *load factor* (*далее - степень загруженности*). Грамотно подобранное значение для *степени загруженности* - залог эффективной работы *хеш-таблицы*. Проблема лишь в том, что для каждой такой таблицы это значение является индивидуальным (*зависит от реализации*). Это значение можно найти путем анализа структуры или же тестированием на разных значениях. Этот тест посвящён подбору значения для *степени загруженности.*

Взглянем на получившееся графики. *"Скачки"* на графиках показывают тот момент, когда загруженность массива превысила допустимое значение. Сразу же после расширения массива **поиск** ключей происходит быстрее (График 4.2.2).

Значения .99, .88 сразу отпадают из списка подходящих - плохая идея ждать, пока место в массиве совсем закончится. На графиках видно, что когда словарь сильно загружен, то быстродействие снижается. И это логично, ведь чем меньше свободного места в массиве, тем больше возникает коллизий.

Значение .72 (близкое к хеш-таблицам в *Java*) показывает неплохие результаты - достаточно быстрое время **добавления** (относительно других значений) и **поиска** ключей. Значение *степени загруженности*, близкое к 3/4 часто используется в качестве стандартного. Например 2/3 в встроенном словаре *Python,* 3/4 в *Java.*

Значения .52 и .26 хорошо показывают себя на **поиске** ключа, однако из-за слишком частого расширения словаря время на **добавление** заметно увеличивается.

Значение .4 является наиболее подходящим для данной реализации *хеш-таблицы*. Данный параметр показал наилучшие результаты как при **добавлении**, так и при **поиске** ключа.

Приложение

- *dictionaries.py* - *реализации словарей*
- *generate_keys.py* - *генератор ключей*
- *tester.py* - *модуль для тестирования*
- *data_analyzer.py* - *модуль для анализа результатов (преобразует в формат .csv)*