**Отчет по тестированию словарей.**

Задача

Провести тестирование по времени операций(set, get, del, keys, values, items) и объему памяти для словарей, основанных на

* Линейном поиске
* Бинарном поиске
* Хэш-таблице
* Бинарном дереве
* Сбалансированном дереве

и встроенного dict.

Входные данные

Тестирование производилось на заранее сгенерированных данных, являющихся списками чисел размером 10 000, 25 000, 50 000, 100 000, 200 000 элементов. Каждый набор был неупорядоченным.

Параметры вычислительного узла

* Процессор Intel Core i7-3517U
* ОЗУ 8 ГБ
* Windows 8.1 64 bit

Тестируемые алгоритмы

**Линейный поиск** — алгоритм нахождения заданного значения произвольной функции на некотором отрезке. Поиск значения функции осуществляется простым сравнением очередного рассматриваемого и, если значения совпадают, то поиск считается завершённым. O(n)

**Бинарный поиск** - алгоритм поиска элемента в отсортированном массиве, использующий дробление массива на половины. О(log n)

* Определение значения элемента в середине структуры данных. Полученное значение сравнивается с ключом.
* Если ключ меньше значения середины, то поиск осуществляется в первой половине элементов, иначе — во второй.
* Поиск сводится к тому, что вновь определяется значение серединного элемента в выбранной половине и сравнивается с ключом.
* Процесс продолжается до тех пор, пока не будет найден элемент со значением ключа или не станет пустым интервал для поиска.

**Хэш-таблица** — это структура данных реализующая интерфейс ассоциативного массива, а именно, она позволяет хранить пары (ключ, значение) и выполнять три операции: операцию добавления новой пары, операцию поиска и операцию удаления пары по ключу.

Хэш-таблица имеет длину и список списков, называемых корзинами.

Заполнение хэш-таблицы: сначала считается хэш от ключа(число), затем берется остаток от длины и в корзину с этим индексом добавляется пара (ключ, значение).

Поиск элемента по ключу: считается хэш, происходит линейный поиск внутри корзины с этим индексом(сравниваются ключи).

Когда в таблице становится много элементов, повышается вероятность коллизии, поэтому происходит увеличение таблицы с пересчетом индексов элементов в таблице.

**Двоичное дерево поиска** — это двоичное дерево, для которого выполняются следующие дополнительные условия:

* Оба поддерева — левое и правое — являются двоичными 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 *больше либо равны*, нежели значение ключа данных самого узла X.

Поиск элемента. Если дерево пусто, сообщить, что узел не найден, и остановиться. Иначе сравнить K со значением ключа корневого узла X.

* Если K=X, выдать ссылку на этот узел и остановиться.
* Если K>X, рекурсивно искать ключ K в правом поддереве Т.
* Если K<X, рекурсивно искать ключ K в левом поддереве Т.

**АВЛ-дерево** — сбалансированное по высоте двоичное дерево поиска: для каждой его вершины высота её двух поддеревьев различается не более чем на 1.

Балансировкой вершины называется операция, которая в случае разницы высот левого и правого поддеревьев = 2, изменяет связи предок-потомок в поддереве данной вершины так, что разница становится <= 1, иначе ничего не меняет. Указанный результат получается вращениями поддерева данной вершины. После каждого добавления или удаления происходит балансировка.

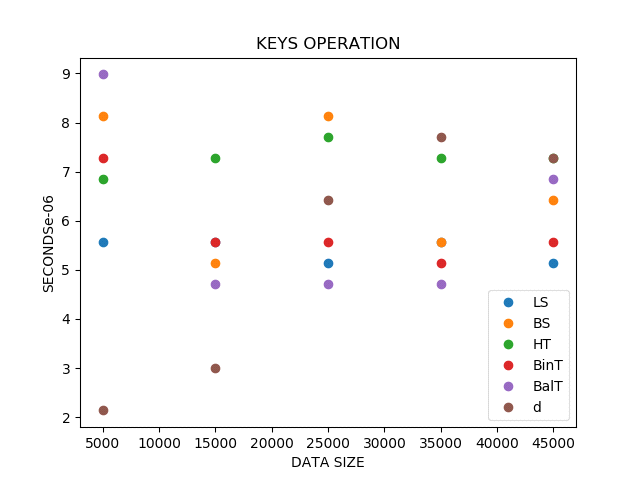
**Результаты измерений**

Операции KEYS, VALUES, ITEMS

**Все три операции реализованы одинокого во всех словарях: возвращают ссылку на список с нужными элементами (хранимую в поле класса).**

**Некоторые значения настолько совпадают, что одна из точек закрывает полностью другую. Различия в значениях малы (порядка 1e-06), связаны с погрешностью в измерении.**

**Показатели не имеют монотонной зависимости, потому что время выполнения этих операций не зависит от объема данных, а только от самой скорости выполнения команд на компьюторе.**



LinearSearchDictionary: {5000: 5.559000001653658e-06, 15000**: 5.558999987442803e-06**, 25000: 5.131000079927617e-06, 35000: **5.560000317927916e-06**, 45000: 5.130999852553941e-06},

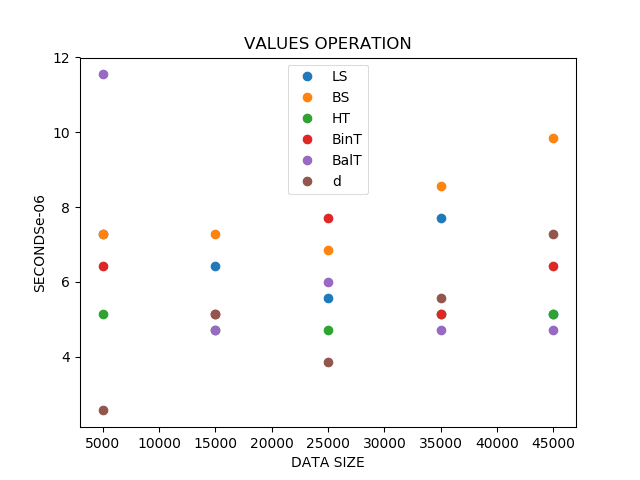
BinarySearchDictionary: {5000: 8.124999993697202e-06, 15000: 5.131000079927617e-06, 25000: 8.125000022118911e-06, 35000: **5.559000783250667e-06**, 45000: 6.415000825654715e-06},

HashTableDictionary: {5000: 6.842000004780857e-06, 15000: 7.2690002070885384e-06, 25000: 7.697000000916887e-06, 35000: 7.26999860489741e-06, 45000: **7.2689981607254595e-06**},

BinaryTreeDictionary: {5000: 7.269999997561172e-06, 15000: **5.558999873755965e-06**, 25000: 5.558999873755965e-06, 35000: 5.131998477736488e-06, 45000: 5.559002602240071e-06},

BalancedTreeDictionary: {5000: 8.98100000767954e-06, 15000: 4.704000048150192e-06, 25000: 4.7040002755238675e-06, 35000: 4.703999366029166e-06, 45000: 6.843001756351441e-06},

dict: {5000: 2.1379999850523745e-06, 15000: 2.992999952766695e-06, 25000: 6.4149999161600135e-06, 35000: 7.697999535594136e-06, 45000: **7.270002242876217e-06**}



LinearSearchDictionary'>: {5000: 7**.2700000011138854e-06**, 15000: 6.413999983578833e-06, 25000: 5.558999873755965e-06, 35000: 7.697000000916887e-06, 45000: **5.131998477736488e-06**},

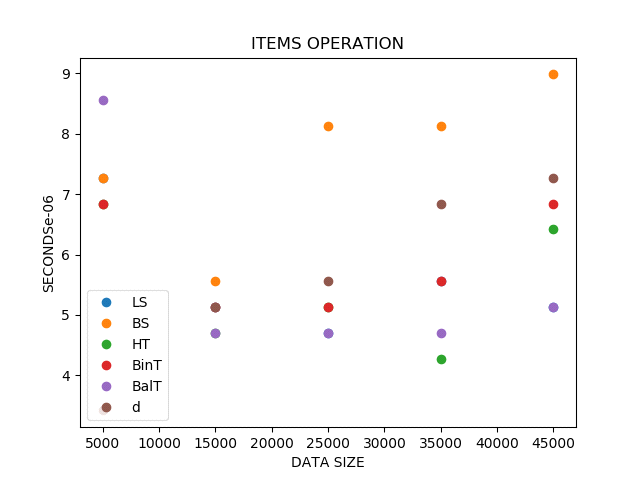
BinarySearchDictionary'>: {5000: **7.269999997561172e-06**, 15000: 7.269999969139462e-06, 25000: 6.842000402684789e-06, 35000: 8.552999133826233e-06, 45000: 9.835999662755057e-06},

HashTableDictionary'>: {5000: 5.1319999982979425e-06, 15000: **4.7039998207765166e-06**, 25000: 4.703999366029166e-06, 35000: **5.1320002967258915e-06**, 45000: **5.131998477736488e-06**},

BinaryTreeDictionary'>: {5000: 6.413999997789688e-06, 15000: **5.130999852553941e-06**, 25000: 7.697000000916887e-06, 35000: **5.130999852553941e-0**6, 45000: 6.414000381482765e-06},

BalancedTreeDictionary'>: {5000: 1.154599999608763e-05, 15000: **4.704000048150192e-06**, 25000: 5.9869998949579895e-06, 35000: 4.703999366029166e-06, 45000: 4.704001185018569e-06},

dict'>: {5000: 2.5660000062543986e-06, 15000: **5.131999841978541e-06**, 25000: 3.848999767797068e-06, 35000: 5.559000783250667e-06, 45000: 7.26999860489741e-06}



LinearSearchDictionary'>: {5000: **7.269999997561172e-06**, 15000: **5.130999966240779e-06**, 25000: **4.704000048150192e-06**, 35000: 5.558999873755965e-06, 45000: **5.1320002967258915e-06**},

BinarySearchDictionary'>: {5000: **7.269999997561172e-06**, 15000: 5.558999987442803e-06, 25000: 8.125000022118911e-06, 35000: 8.125000022118911e-06, 45000: 8.980001439340413e-06},

HashTableDictionary'>: {5000: **6.842000004780857e-06**, 15000**: 4.704000048150192e-06**, 25000: **5.1320002967258915e-06**, 35000: 4.276000254321843e-06, 45000: 6.414997187675908e-06},

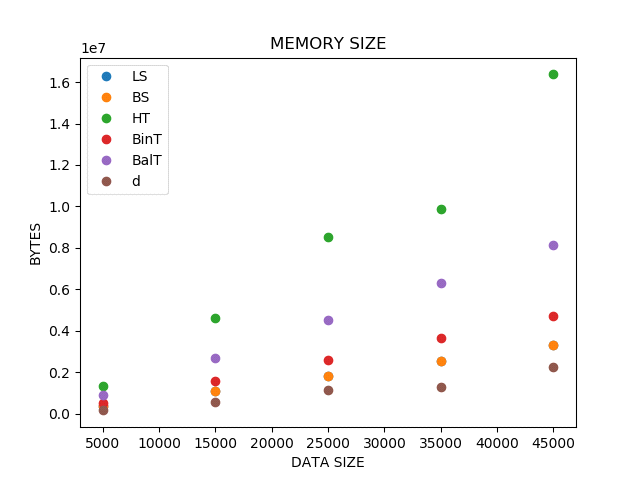
BinaryTreeDictionary'>: {5000: **6.841999990570002e-06**, 15000: **5.131000079927617e-06**, 25000: **5.1320002967258915e-06**, 35000: 5.559999408433214e-06, 45000: 6.842001312179491e-06},

BalancedTreeDictionary'>: {5000: 8.551999997052917e-06, 15000**: 4.7039998207765166e-06**, 25000: **4.7040002755238675e-06**, 35000: 4.704001185018569e-06, 45000: **5.132002115715295e-06**},

dict'>: {5000: 3.4210000023904286e-06, 15000: **5.132000069352216e-06**, 25000: 5.558999873755965e-06, 35000: 6.841999493190087e-06, 45000: 7.26999860489741e-06}

**Затраты по памяти**

График показывает объем памяти, выделенный на словарь, для каждого из размеров наборов данных. **Значения на оси Y нужно умножить на 10^7(верхн. лев. угол)**



*Точные значения:*

LinearSearchDictionary: {5000: 364600, 15000: 1087420, 25000: 1800520, 35000: 2537044, 45000: 3290900},

BinarySearchDictionary: {5000: 364600, 15000: 1087420, 25000: 1800520, 35000: 2537044, 45000: 3290900},

HashTableDictionar : {5000: 1322200, 15000: 4597180, 25000: 8499120, 35000: 9885644, 45000: 16371184},

BinaryTreeDictionary: {5000: 524608, 15000: 1567428, 25000: 2600528, 35000: 3657052, 45000: 4730908},

BalancedTreeDictionary: {5000: 903656, 15000: 2704232, 25000: 4494920, 35000: 6309162, 45000: 8141030},

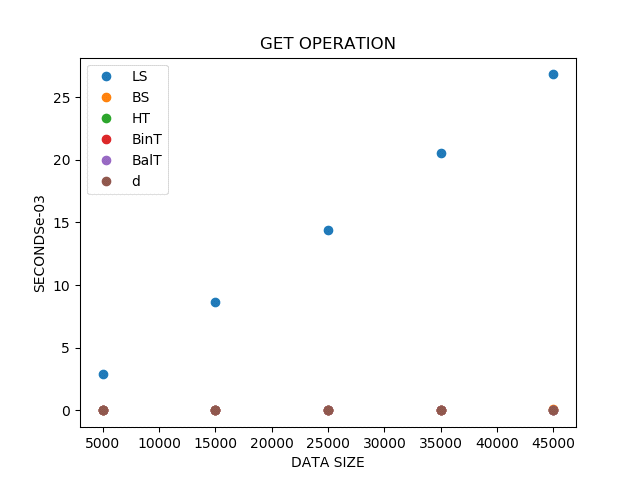
dict: {5000: 151982, 15000: 537742, 25000: 1136490, 35000: 1280954, 45000: 2227390}

По числам можно заметить, что размер LinearSearchDictionary и BinarySearchDictionary точно совпдет, и это отвечает тому, что структура у них полностью одинаковая.

Анализ показывает, что для реализаций, где для ускорения доступа используются дополнительные структуры(BinaryTreeDictionary , BalancedTreeDictionary, HashTableDictionary), требуется больший объем памяти.

**Операция GET**

График отражает соответствие объема входных данных и среднего времени доступа к одному элементу (время доступа всех, деленное на число элементов).



Тоные значения:

LinearSearchDictionary'>: {5000: 0.0028006566778000005, 15000: 0.008809486874, 25000: 0.014549870417799997, 35000: 0.020184968366571437, 45000: 0.026785136992199983},

BinarySearchDictionary'>: {5000: 5.075439240000037e-05, 15000: 6.471904179999607e-05, 25000: 6.712130064000121e-05, 35000: 7.118381708569359e-05, 45000: 7.538353951111073e-05},

HashTableDictionary'>: {5000: 6.7450151999992155e-06, 15000: 8.05831360000108e-06, 25000: 6.165312079992873e-06, 35000: 6.659317028580907e-06, 45000: 7.018673711111963e-06},

BinaryTreeDictionary'>: {5000: 4.631749899999989e-05, 15000: 5.332667933333444e-05, 25000: 5.5309467280003444e-05, 35000: 6.379821548569972e-05, 45000: 6.413988026668247e-05},

BalancedTreeDictionary'>: {5000: 3.6928411200000254e-05, 15000: 4.2504412466670754e-05, 25000: 4.5086010439990786e-05, 35000: 4.4077798942868994e-05, 45000: 4.706712582224605e-05

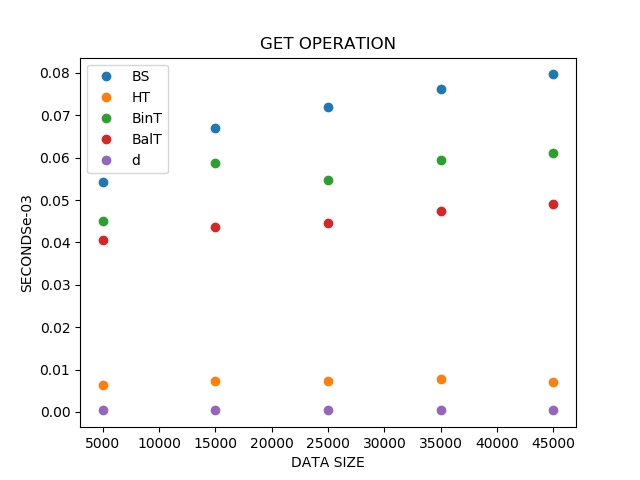
dict'>: {5000: 3.474113999999418e-07, 15000: 4.248419999991408e-07, 25000: 5.218868399970234e-07, 35000: 4.68688914276884e-07, 45000: 4.79104244449344e-07}

Значения для словарей, кроме LinearSearchDictionary, в экспоненциальном представлении имеют степени -5,-6,-7. Поэтому на графике точки двух цветов, остальные по сравнению с LinearSearchDictionary, настолько малы, что приближаются к нулю.

***Анализ***

Не стоит использовать LinearSearchDictionary, если нужен доступ к элементам в случайном порядке. Для этого можно выбрать любой другой из использованных словарей. Но LinearSearchDictionary, довольно мало занимает по памяти (см. Затраты по памяти), поэтому его можно использовать, если надо записать, а потом пройтись по всем элементам.

***Повторный анализ, только для быстрых алгоритмов***

******

По графику можно увидеть, что алгоритмы, основанные на бинарном поиске, проигрывают хэш-таблице в быстроте поиска элемента.

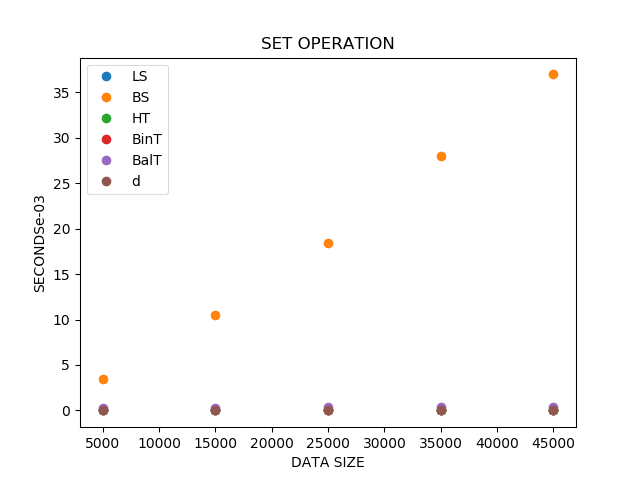
Результат

Чтобы иметь наиболее быстрый доступ к элементам стоит использовать хэш-таблицу(полагаем, что хэш-функция дает мало коллизий), если нет особой причины для использования бинарного поиска(например, используемый алгоритм).

Если же необходимо использовать бинарный поиск, то наиболее быстрым вариантом является сбалансированное дерево.

**Операция SET**

График отражает соответствие объема входных данных и среднего времени добавления одного элемента (время добавления всех, деленное на число элементов).



LinearSearchDictionary'>: {5000: 9.867698000000047e-06, 15000: 1.0270673533333744e-05, 25000: 6.840412279998418e-06, 35000: 7.359882342851571e-06, 45000: 7.230267955573152e-06},

BinarySearchDictionary'>: {5000: 0.0034140202134000006, 15000: 0.010504558798066667, 25000: 0.018333394890119998, 35000: 0.027329122113628583, 45000: 0.03702988453544444},

HashTableDictionary'>: {5000: 2.8723104999997418e-05, 15000: 3.904398293333694e-05, 25000: 4.539632007999899e-05, 35000: 4.0224384228583533e-05, 45000: 5.120475957778075e-05

BinaryTreeDictionary'>: {5000: 5.177430399999991e-05, 15000: 4.718126660000053e-05, 25000: 4.719386188000499e-05, 35000: 4.797807034285922e-05, 45000: 5.3136077266632736e-05},

BalancedTreeDictionary'>: {5000: 0.00031329468640000187, 15000: 0.0003170453667333277, 25000: 0.0003286628937600108, 35000: 0.0003384754756285734, 45000: 0.00034754043322221453},

BinaryTreeDictionary'>: {5000: 7.392113999998174e-07, 15000: 5.472598666680521e-07, 25000: 7.939316800002416e-07, 35000: 5.180503428554012e-07, 45000: 6.628356222386679e-07}

Результаты измерений можно разделить на три группы:

* BinarySearchDictionary'
* BalancedTreeDictionary'
* все остальные

Больше всего затрат при добавлении у BinarySearchDictionary, так как необходимо поддерживать отсортированный порядок.

Дальше идет BalancedTreeDictionary, требующий проверки выполнения правил авл-дерева относительно высот и ребалансировки.

Значения всех остальных сравнительно малы и очень близки по величине.

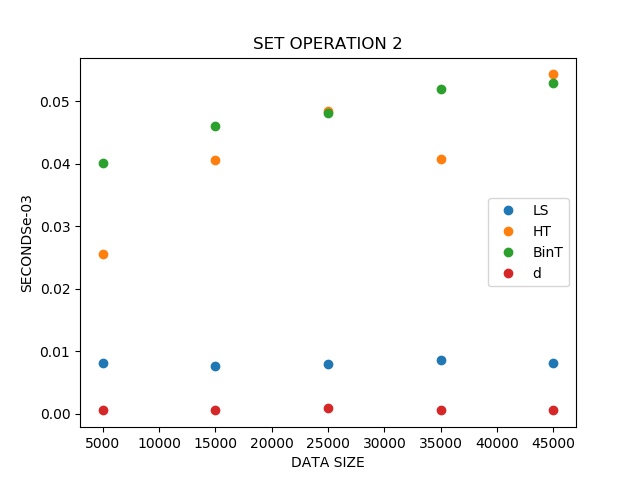
**Анализ**

Если не важно, как долго будут добавляться элементы, а так же имеет значение занимаемый объем памяти, то есть смысл использовать BinarySearchDictionary.

Если же важна скорость добавления в словарь, лучше использовать

LinearSearchDictionary, HashTableDictionary, BinaryTreeDictionary или BinaryTreeDictionary.

***Повторный анализ, только для быстрых алгоритмов***

******

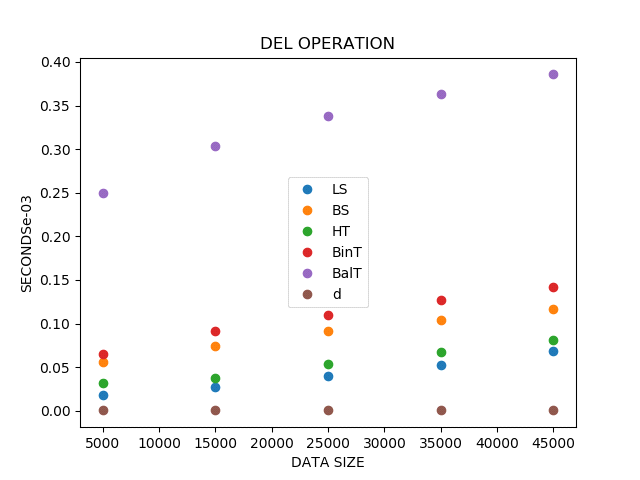
***По графику можно заметить, что самыми быстрыми являются встроенный словарь dict и реализация LinearLearchDictionary, а словари на бинарном дереве и хэш-таблице чуть отстают.***

***Результат***

***Лучшими вариантами с точки зрения скорости добавления элементов являются словарь на основе списка с линейным поиском и на основе хэш-таблице (с быстрой хэш-функцией).***

**Операция DEL**

График отражает соответствие объема входных данных и среднего времени удаления одного элемента (время удаления всех, деленное на число элементов).



Точные значения:

LinearSearchDictionary: {5000: 1.6310203799999813e-05, 15000: 2.661571526666838e-05, 25000: 3.837633500000265e-05, 35000: 5.2359028914283306e-05, 45000: 6.89266082666816e-05

BinarySearchDictionary: {5000: 6.146667139999807e-05, 15000: 7.526727953333345e-05, 25000: 9.03332395599864e-05, 35000: 0.00010508544594285922, 45000: 0.00011818199442220955},

HashTableDictionary: {5000: 2.628010659999802e-05, 15000: 3.649196600000172e-05, 25000: 5.371790916000464e-05, 35000: 7.137767054287646e-05, 45000: 8.442016042220834e-05},

BinaryTreeDictionary: {5000: 7.498159019999946e-05, 15000: 8.974837053333432e-05, 25000: 0.00010801991756001371, 35000: 0.00013092420788569662, 45000: 0.00014234728957777002},

BalancedTreeDictionary: {5000: 0.00027740902259999983, 15000: 0.00030903027293333404, 25000: 0.0003527385716400022, 35000: 0.0003586456062857126, 45000: 0.00037281436011111916},

dict: {5000: 3.84957800000052e-07, 15000: 3.9644700000280866e-07, 25000: 3.9231315999131767e-07, 35000: 4.065717714443703e-07, 45000: 5.084400444401479e-07}

Медленнее всех в среднем BalancedTreeDictionary, это связано с ребалансировкой.

**Анализ**

BalancedTreeDictionary можно использовать, если не слишком важна скорость удаления (например, редко удаляем).

Но в целом, как видно из графика и того, что точек везде 6, это операция работает за примерно одинаковое время.