Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет
«Высшая школа экономики»
Факультет математики

Гущин Антон Ильич

Методы разрешения хеш-коллизий. Hopscotch hashing

Курсовая работа студента 3 курса
образовательной программы бакалавриата «Математика»

Научный руководитель:
Парфёнов Денис Васильевич

Москва 2024

1 Хеш-таблицы

Хеш-таблица[1] - это структура данных, позволяющая хранить пары "ключ-значение" и выполнять (за $O(1)$ амортизированного времени) три операции: добавление новой пары, удаление и поиск пары по ключу. Ключи обычно хранятся в массиве, а поиск идёт с помощью вычисления некоторой хеш-функции от ключа. Хеш-коллизия возникает, когда разные ключи имеют одно и то же значение хеш-функции: они негативно влияют на производительность, поэтому способ их разрешения - важная составляющая любой хеш-таблицы.

2 Устройство hopscotch хеш-таблицы

2.1 Функции

Hopscotch хеш-таблица имеет следующие функции:

- Открытые (public):
  1. **add(x):** добавить в таблицу элемент x
  2. **remove(x):** удалить из таблицы элемент x
  3. **contains(x):** проверить, есть ли в таблице элемент x
- Закрытые (private):
  1. **resize():** попробовать увеличить размер таблицы
  2. **tryadd(x):** проверить, можно ли добавить в таблицу элемент x, не увеличивая размер таблицы

2.2 Переменные и параметры

Контейнеры: 2 массива - **values** (ключи) и **bitmaps** (битовые маски вёдер - показывают, в каких из следующих HOP_RANGE ячеек лежат ключи, относящиеся к данному ведру. Ведро - это индекс, в который хешируется ключ: вычисляется по формуле $mod(hash(x), size(values))$). Каждый ключ должен находиться на расстоянии менее HOP_RANGE вправо от своего ведра.

Параметры: HOP_RANGE - максимальное допустимое ориентированное расстояние от ведра до всех ключей в нём + 1

Алгоритм **tryadd(x)**:

1. Вычислить индекс ведра по формуле $ind = mod(hash(x),\ size(values))$
2. Найти ближайшую справа свободную ячейку в **values** (переходя в начало, если дошли до конца). Если не нашли - завершить работу с ошибкой
3. Если расстояние от ведра до свободной ячейки при движении вправо меньше HOP_RANGE, то добавить в эту ячейку x (и добавить соответствующий бит в битовую маску ведра x), после чего завершить работу с успехом
4. В противном случае попробовать осуществить *прыжок*: идя по вёдрам слева направо, начиная от ведра на расстоянии $HOP\_RANGE - 1$ влево от свободной ячейки, проверять, есть ли в этом ведре число, находящееся в ячейке левее свободной ячейки. Если есть - то перенести его в свободную ячейку и повторить *прыжок*, если расстояние всё ещё больше HOP_RANGE, иначе перейти к шагу 3. Если нет - то прекратить работу с ошибкой: добавить элемент не получилось

Алгоритм **resize()**:

1. Создать массив с размером, равным удвоенному размеру **values**
2. Проходя от начала **values** к концу, вставлять каждый элемент в новый массив с помощью **tryadd(x)**. В случае неудачи прибавить к размеру массива $size(values)$ и повторить этот шаг заново.

Алгоритм **add(x)**:

1. Вызвать **tryadd(x)**. В случае удачи завершить работу
2. В случае неудачи вызвать **resize()** и перейти к шагу 1

Алгоритм **contains(x)**:

1. Найти ведро, соответствующее ключу x (так же, как в **tryadd(x)**).
2. Рассмотреть минимальный бит в битовой маске этого ведра, проверить ячейку **values**, соответствующую ему. Если в ней лежит ключ x - то завершить работу успешно, иначе повторить этот шаг для следующего минимального бита битовой маски
3. Если в битовой маске не осталось битов - то числа x в таблице нет, завершить работу с ошибкой

Алгоритм **remove(x)**:

1. Осуществить шаги 1-2 **contains(x)**. Если ключ x в **values** найден - то удалить его оттуда и заменить соответствующий бит в битовой маске ведра для x. Иначе (если x в таблице нет) завершить работу с ошибкой

3 Оценка времени работы функций

3.1 Необходимая теория

Здесь мы будем считать известной всю информацию о хешировании с открытой адресацией (open-addressing[2]), а также предполагаем, что используемая хеш-функция равномерна. Считаем, что размер таблицы лежит между 10^3 и 10^9, а $HOP\_RANGE = 32$. Обозначения: n - количество ключей в таблице, m - размер таблицы.

3.2 Численные оценки

Теорема 1: В таблице с load factor $= \alpha$ мат.ожидание ориентированного расстояния от ведра до ячейки с ключом в этом ведре не более $\frac{1}{2}(1 + \frac{1}{(1-\alpha)^2})$
Доказательство: В силу того, что прыжки не меняют сумму расстояний от вёдер до ячеек, мат.ожидание этого расстояния равно мат.ожиданию такого расстояния для open-addressing таблицы. А для неё это уже доказано[3] □

Лемма 1: Если мат.ожидание расстояния от ведра до ячейки не более E, то вероятность $p(E, X)$ того, что ячейка с ключом будет на расстоянии хотя бы X от ведра, не более чем $\frac{E}{X}$
Доказательство: Расстояние не менее 0, поэтому если вероятность равна p, то мат.ожидание не

менее $p \times X + (1-p) \times 0 = p \times X$. С другой стороны, оно не больше E. Значит, $p(E, X) \leq \frac{E}{X}$ $\square$

Лемма 2: Пусть мат.ожидание расстояние от ведра до ячейки не более E, и распределения расстояния от ведра для ячейки по всем вёдрам независимы. Тогда вероятность неудачи при прыжке не более $\frac{E^{31}}{31!}$
Доказательство: Прыжок неудачен, если и только если ячейка на 1 слева от свободной удалена от своего ведра на 31 вправо, ячейка на 2 слева удалена от своего ведра на 30 или 31 вправо, ..., ячейка на 31 слева удалена на 1, 2, 3, ..., 30 или 31 вправо. Первое событие происходит с вероятностью $p(E, 31)$, второе с вероятностью $p(E, 30)$ и так далее. Из независимости получаем итоговую вероятность равной $p(E, 31) \times ... \times p(E, 1) \leq \frac{E}{31} \times ... \times \frac{E}{1} = \frac{E^{31}}{31!}$ $\square$

Следствие 1: Вероятность неудачи при **resize()** в среднем не более 10^{-19}
Доказательство: Пусть размер таблицы m, а добавляется туда $n \leq \frac{m}{2}$ элементов. Заметим, что при добавлении первого элемента load factor = 0, при добавлении второго load factor = $\frac{1}{m}$ и т.д., вплоть до $\frac{n-1}{m}$. Тогда мат.ожидание суммы расстояний $\overline{E}$ от ячейки до ведра в конце равно сумме мат.ожиданий расстояний от каждой ячейки до ведра при суммировании в порядке добавления. Применив **Теорему 1** к каждому слагаемому и раскрыв скобки, получим $\overline{E} \leq \frac{n}{2} + \frac{1}{2} \times (\frac{1}{(1-\frac{0}{m})^2} + \frac{1}{(1-\frac{1}{m})^2} + ... + \frac{1}{(1-\frac{n-1}{m})^2}) = \frac{n}{2} + \frac{m^2}{2} \times (\frac{1}{m^2} + \frac{1}{(m-1)^2} + ... + \frac{1}{(m-n+1)^2})$. Заметим, что сумма S дробей в скобках - это в точности нижняя частичная сумма $\int_{m-n}^{m} \frac{1}{x^2} dx$. Тогда $S \leq -\frac{1}{x}|_{m-n}^{m} = \frac{n}{m(m-n)} \leq \frac{n}{m(m-m/2)} = \frac{2n}{m^2}$. Значит, $\overline{E} \leq \frac{n}{2} + \frac{m^2}{2} \times \frac{2n}{m^2} = \frac{n}{2} + n = 1.5n$. При этом матожидание расстояния от ячейки до ведра $E = \frac{\overline{E}}{n} \leq 1.5$, и по **Лемме 2** вероятность неудачи при одном прыжке не более $\frac{1.5^{31}}{31!} \approx 3.5 \times 10^{-29}$. Заметим, что поскольку на каждый элемент мы делаем количество прыжков не большее расстояния от первой свободной ячейки до его ведра (т.к. каждый прыжок приближает хотя бы на 1), то суммарно мы сделали прыжков в среднем не больше $1.5n \leq 1.5 \times 10^9$, значит, вероятность провалиться хотя бы в одном не больше $1.5 \times 10^9 \times 3.5 \times 10^{-29} = 5.25 \times 10^{-20} \leq 10^{-19}$ $\square$

Следствие 2: При load factor ≤ 0.72 вероятность вызова **resize()** при использовании **add(x)** в среднем не больше 0.53×10^{-9}
Доказательство: по **Теореме 1** получим, что $E \leq 6.88$. В доказательстве **Леммы 2** заменим $\frac{E}{6}, ..., \frac{E}{1}$ на 1 (поскольку $p(E, 6), ..., p(E, 1)$ - вероятности, они не могут быть больше единицы) и вычислением получим, что вероятность неудачи при одном прыжке не больше 7.63×10^{-11}. Прыжков в среднем не больше $E \leq 6.88$, значит, вероятность неудачи в среднем не больше $7.64 \times 6.88 \times 10^{-11} \leq 0.53 \times 10^{-9}$ $\square$

Таким образом, таблица хорошо работает при загрузке до 0.72 (т.к. размер таблицы не больше 10^9, вызов resize() при load factor ≤ 0.72 будет происходить редко).

3.3 Асимптотические оценки

Теорема 2: При load factor $\leq \overline{\alpha} < 1$ **tryadd(x)** выполняется за $O(1)$ амортизированного времени и требует $O(1)$ доп.памяти в худшем случае
Доказательство: Из **Теоремы 1** получим верхнюю оценку на матожидание расстояния от ведра до ячейки с ключом как функцию от $\overline{\alpha}$. Поскольку $\overline{\alpha}$ фиксировано и отделено от 1, эта верхняя оценка является $O(1)$. В таком случае 2 шаг **tryadd(x)** выполняется за $O(1)$ времени в среднем. Поскольку каждый прыжок уменьшает текущее расстояние хотя бы на 1, а для проверки возможности прыжка необходимо сделать не более $HOP\_RANGE = 32 = O(1)$ проверок, 4 шаг делается в среднем за время, асимптотически не большее времени 2 шага. Очевидно, что шаги 1 и 3 делаются за $O(1)$ времени в худшем случае, а также каждый шаг 1-4 требует $O(1)$ памяти в худшем случае $\square$

Теорема 3: resize() выполняется за $O(m)$ амортизированного времени и требует всегда $O(m)$ дополнительной памяти
Доказательство: Заметим, что если первый вызов **resize()** не осуществляет повторных вызовов, то он делает $n \leq m$ вызовов **tryadd(x)** и m проверок на существование элемента в ячейке. Вторая часть выполняется за m элементарных операций в худшем случае; для оценки первой части восполь-

зуемся доказательством **Теоремы 2**, подставив туда $\overline{\alpha} = 0.5$. Получим, что в итоге 2 шаг **tryadd(x)** делается в среднем не более чем за $2.5 + 1 = 3.5$ элементарных операций (проверить ячейку-ведро и еще в среднем $\overline{\alpha}$ других ячеек), 4 шаг делается не более чем за $(31 + 4) \times 3.5 = 122.5$ элементарных операций (31 на поиск ячейки, 4 на перенос значения и замену бита, не более чем 3.5 прыжка в среднем), шаг 1 делается за 2 элементарных операции, шаг 3 делается за 2 элементарных операции. Тогда весь **resize()** в случае успеха занимает не более чем $m+3.5n+122.5n+2+2 = m+126n+4 < 127n+m$ элементарных операций
Заметим, что из **Следствия 1** вероятность повторного вызова **resize()** меньше $\frac{1}{2\times(127n+m)}$, поэтому вместе с повторными вызовами **resize()** занимает не более чем $(m + 127n) \times 2 = 2m + 254n = O(n + m) = O(m)$ амортизированного времени. Поскольку шаг 1 **resize()** требует $O(m)$ дополнительной памяти всегда (т.к. был массив размера m, создаётся массив размера 2m, а старый будет потерян), а каждый **tryadd(x)** требует $O(1)$ доп.памяти худшего случая, но они вызываются последовательно, поэтому суммарно они требуют тоже $O(1)$ памяти худшего случая, поэтому итого требуется $O(m)$ памяти всегда □

Теорема 4: При load factor ≤ 0.72 **add(x)** выполняется за $O(1)$ амортизированного времени и требует $O(1)$ амортизированной доп.памяти
Доказательство: Из **Теоремы 2** знаем, что первый вызов **tryadd(x)** выполнится за $O(1)$ амортизированного времени и $O(1)$ памяти в худшем случае. Из **Теоремы 3 resize()** выполняется за $O(m)$, поэтому на все вызовы **resize()** суммарно в среднем уйдёт $O(m)$ времени при условии, что хотя бы один вызов произошёл. При этом по **Следствию 2** вероятность вызова **resize()** не более чем $0.53 \times 10^{-9} \leq \frac{0.53}{m}$, значит, все вызовы **resize()** занимают амортизированно $O(1)$ времени, а повторные вызовы **tryadd(x)** занимают в среднем $O(\frac{1}{n})$ времени. Итого амортизированного времени на вызов **add(x)** нужно $O(1) + O(1) + O(\frac{1}{n}) = O(1)$
Аналогично, памяти на первый вызов **tryadd(x)** нужно $O(1)$ в худшем случае, на все вызовы **resize()** - $O(1)$ в среднем, и на повторные вызовы **tryadd(x)** - $O(\frac{1}{n})$ в среднем, тогда амортизированно требуется $O(1) + O(1) + O(\frac{1}{n}) = O(1)$ доп.памяти □

Теорема 5: contains(x) выполняется за $O(1)$ времени и $O(1)$ доп.памяти в худшем случае
Доказательство: Шаг 1 выполняется за константное время и доп.память. Дальше нам необходимо проверить не более чем 32 ячейки. Каждая проверка осуществляется за не более чем константное время и доп.память в худшем случае, поэтому все проверки осуществятся за $O(1)$ времени и доп.памяти □

Теорема 6: remove(x) выполняется за $O(1)$ времени и $O(1)$ доп.памяти в худшем случае
Доказательство: по **Теореме 5** вызов **contains(x)** выполняется за $O(1)$ времени и $O(1)$ доп.памяти в худшем случае. Удаление числа из массива и замена бита в битовой маске - элементарные действия, поэтому весь **remove(x)** тоже требует $O(1)$ времени и $O(1)$ доп.памяти в худшем случае □

4 Детали реализации

- В качестве хеш-функции используется fnv1a[4]. Она вычисляется за $O(1)$ в худшем случае и равномерна
- Массивы **values** и **bitmaps** изначально заполнены нулями. Мы постоянно поддерживаем индекс ведра, куда попал бы 0, и постоянно помним его битовую маску - это позволяет осуществлять проверку на то, является ли ячейка на самом деле пустой (а не лежащим в таблице нулём), за $O(1)$, не увеличивая асимптотически время выполнения всех функций. Соответственно, "удаление" элемента из массива - это на самом деле замена его на 0, а "добавление" элемента x - замена соответствующего нуля на x
- Также в таблице всегда поддерживается количество занятых ячеек. Это добавляет $O(1)$ ко времени исполнения **tryadd(x)**, **remove(x)** и **resize()**, поэтому не влияет на асимптотику, но позволяет вычислять load factor за $O(1)$ времени и памяти худшего случая
- Для 4 шага **tryadd(x)** проверка происходит так: смотрим на битовую маску очередного

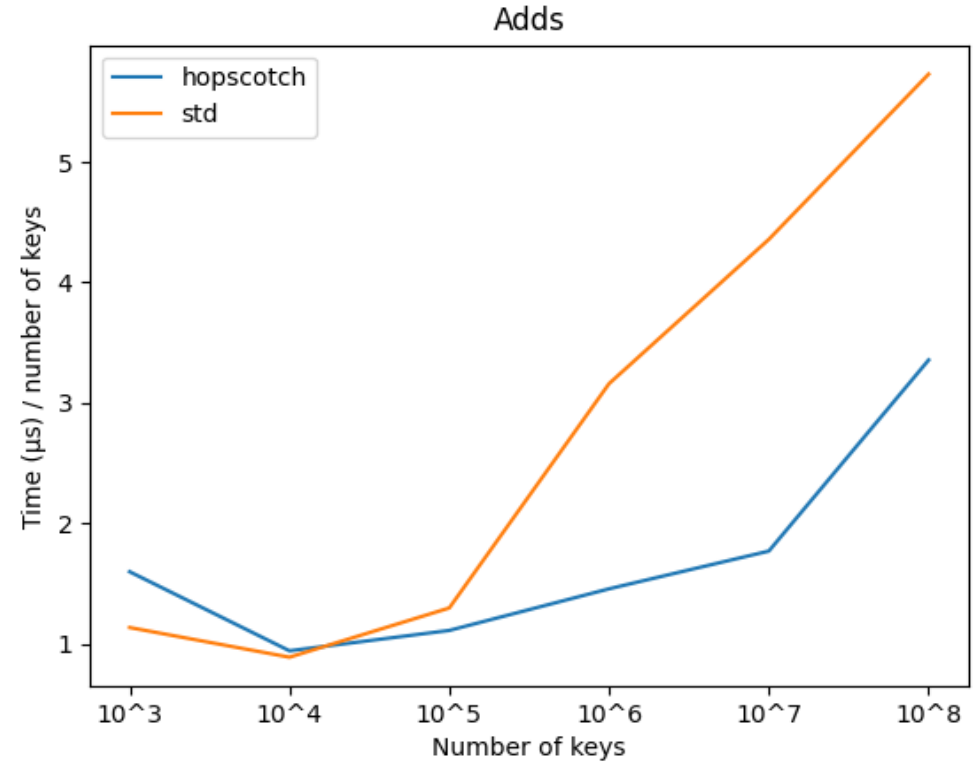
ведра, берём в ней минимальный положительный бит (с помощью инструкции процессора _BitScanForward) и если он соответствует ячейке левее пустой - то прыжок возможен

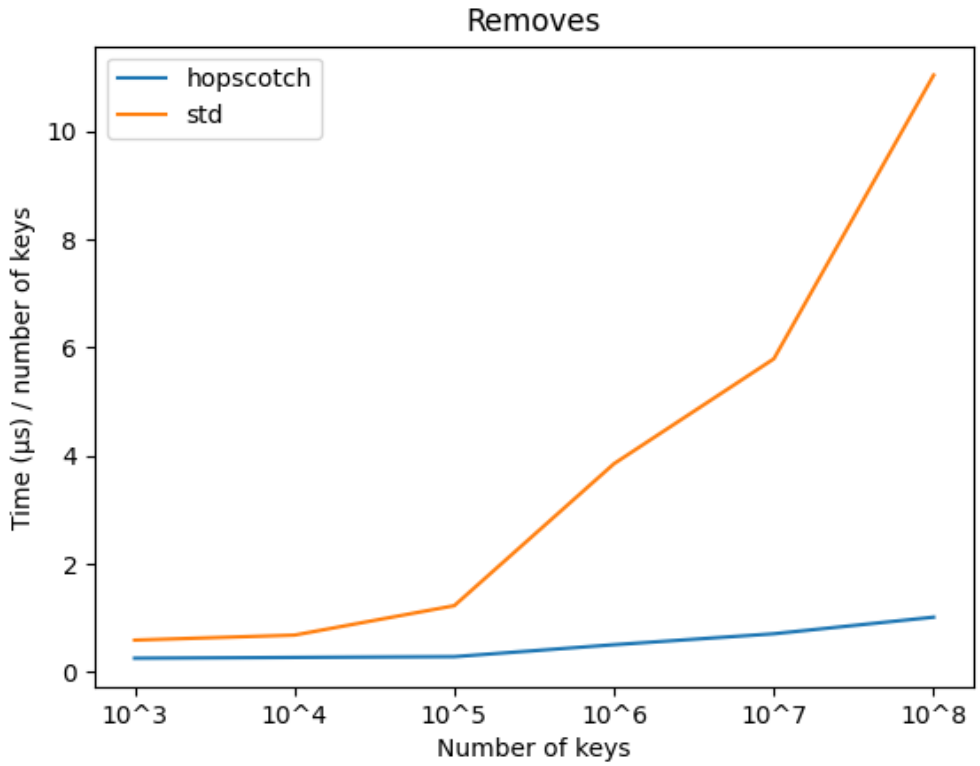
- Дополнительный параметр ADD_RANGE: максимальное расстояние вправо от ведра, на котором мы ищем свободную ячейку в **tryadd(x)** (если не нашли - то завершаем работу сразу). Это приводит к более ранним вызовам **resize()**, но на практике (см. раздел 4) выбор параметра $ADD\_RANGE = 128$ сильно улучшает работу

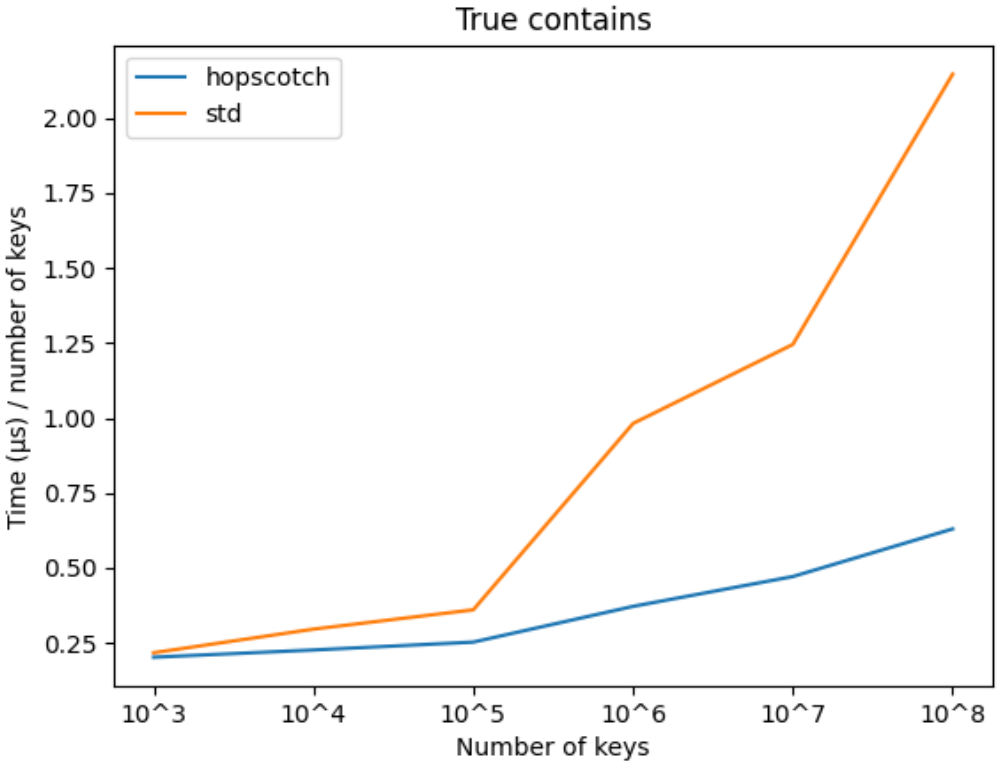
- Дополнительный параметр MAX_TRIES: отвечает за максимальное количество попыток **resize()** и **tryadd(x)** подряд. Если за MAX_TRIES попыток таблица не справилась с операцией, то вместо дальнейших попыток программа завершает работу. Не влияет на скорость работы: добавлено для того, чтобы программа не пыталась делать операцию бесконечно (если у нас в таблице появляется $HOP\_RANGE+1$ одинаковых элементов, то ни при каком размере таблицы алгоритм не сможет вставить туда все эти элементы и без введения MAX_TRIES работал бы бесконечно). Выбор $MAX\_TRIES = 5$ позволяет сделать вероятность ложного срабатывания пренебрежимо малой, и при этом достаточно эффективно завершает работу при верном срабатывании

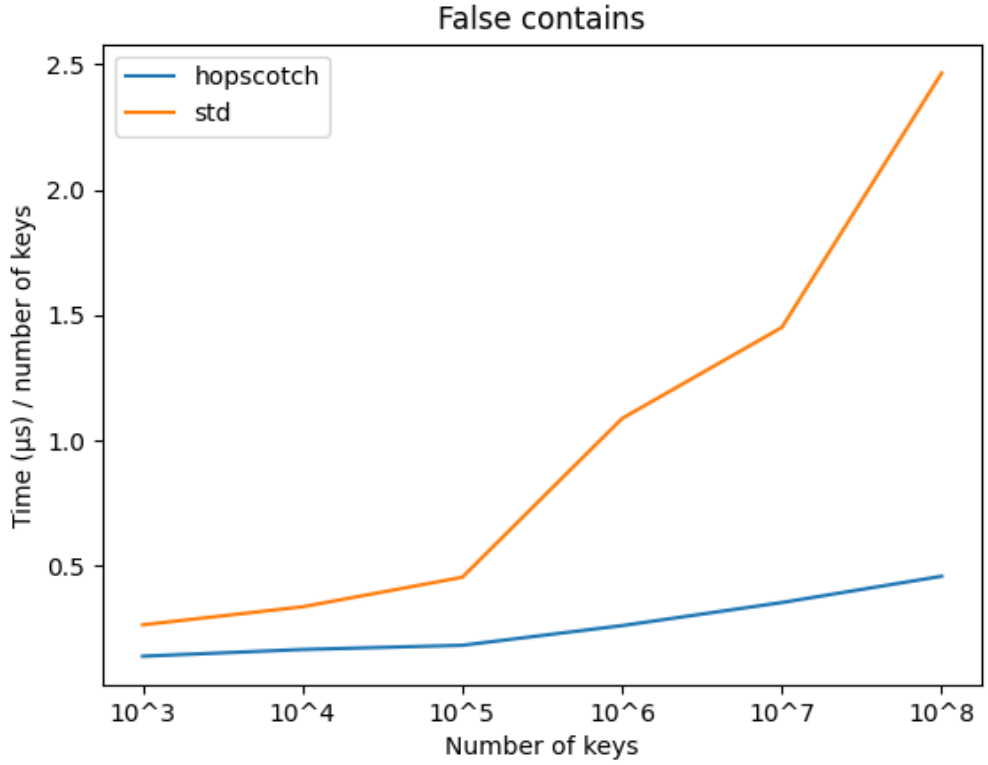
5 Численные эксперименты, сравнение со стандартной хеш-таблицей

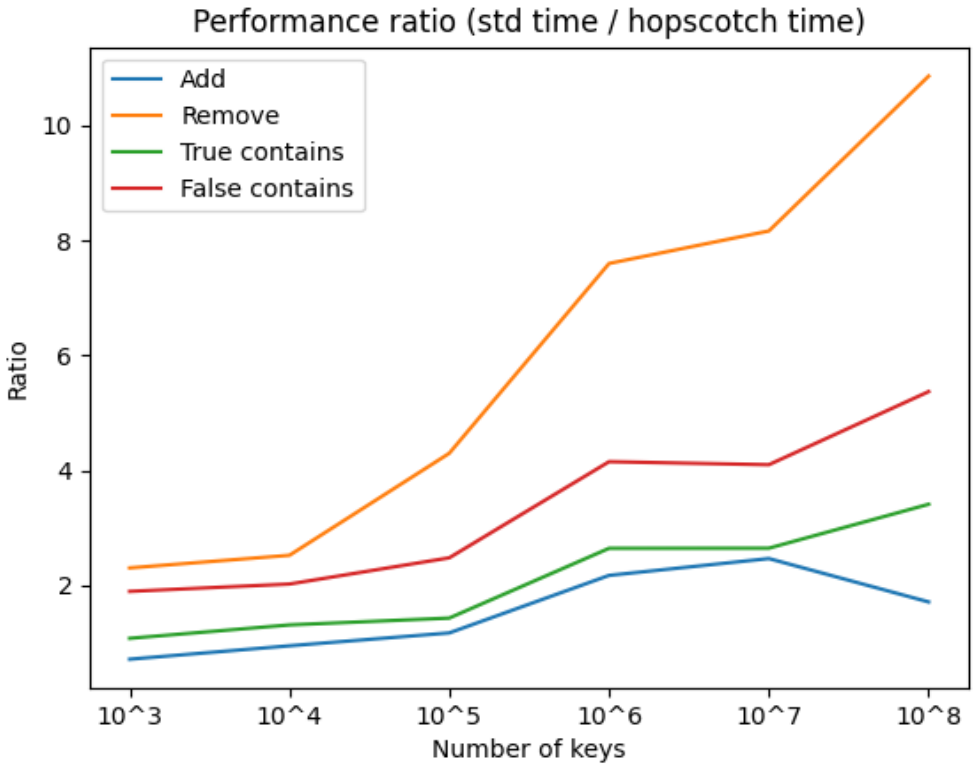
Ради скорости вычислений, тесты таблицы размера 10^n были проведены $15 \times 10^{8-n}$ раз, за исключением 10^9, где тест был проведён 3 раза (на графиках изображены средние результаты тестов). Каждый тест использовал одинаковый случайно сгенерированный набор ключей для hopscotch таблицы (реализованной на C++) и для стандартной std::unordered_multiset, были скомпилированы с одинаковыми настройками компилятора. В тестах **remove(x)** и **contains(x)** ключи были перемешаны в случайном порядке перед тестированием. Тестирование **add(x)** измеряло время на вставку N ключей в таблицу, тестирование **remove(x)** - время на удаление N ключей после вставки, тестирование **contains(x)** - время на N проверок. True contains - вызов **contains(x)**, когда x заведомо есть в таблице; false contains - когда x заведомо нет в таблице.


Adds
hopscotch
std
Time (µs) / number of keys
1
2
3
4
5
10^3
10^4
10^5
10^6
10^7
10^8
Number of keys


Removes
hopscotch
std
Time (µs) / number of keys
0
2
4
6
8
10
10^3
10^4
10^5
10^6
10^7
10^8
Number of keys


True contains
hopscotch
std
2.00
1.75
1.50
1.25
1.00
0.75
0.50
0.25
Time (µs) / number of keys
10^3
10^4
10^5
10^6
10^7
10^8
Number of keys


False contains
hopscotch
std
2.5
2.0
1.5
1.0
0.5
Time (µs) / number of keys
10^3
10^4
10^5
10^6
10^7
10^8
Number of keys



В процессе тестов было выявлено, что тесты на малых размерах (все при 10^3 и **add(x)** при 10^4 элементов) имеют большой разброс
Как видно из тестов, алгоритм hopscotch сильно обгоняет std::unordered_multiset на всех операциях, начиная с 10^5 ключей в таблице, причём разница в скорости увеличивается с количеством ключей.
Полный код хеш-таблицы и тестов находится по ссылке: https://github.com/aiguschin/hopscotch

Список литературы

[1] https://en.wikipedia.org/wiki/Hash_table

[2] https://en.wikipedia.org/wiki/Open_addressing

[3] Knuth, D. E., *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997

[4] https://en.wikipedia.org/wiki/Fowler-Noll-Vo_hash_function

[5] Herlihy, M., Shavit, N., Tzafrir, M., *Hopscotch Hashing (2008)*