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

Факультет математики

Дипломная работа

Optimized Hash Collision Resolution Methods
Оптимизированные методы разрешения хеш-коллизий

Антон Гущин, БМТ213

Научный руководитель:
Парфенов Денис Васильевич,
к.т.н., доцент

Москва 2025

1 Введение

Хеш-таблица [1] - структура данных, хранящая пары «ключ-значение» и позволяющая выполнять поиск и удаление элемента за $O(1)$ времени в худшем случае, а также вставки за $O(1)$ амортизированного времени.
Эта работа будет сфокусирована на реализациях хеш-таблицы от Google с открытым исходным кодом [2]. Их две: sparse- и dense-hash-table.
Далее будет описан теоретический и практический анализ этих двух таблиц, а также сравнение их с текущей «стандартной» реализацией и двумя другими хеш-таблицами, одна из которых была изобретена автором.

2 Sparse-hash-table

2.1 Sparsetable

Sparse-hash-table использует структуру данных под названием **sparsetable** как хранилище пар «ключ-значение». Основная причина использования **sparsetable** вместо другого контейнера - ограничить расход памяти: такой выбор хранилища обеспечивает использование дополнительной памяти в размере всего 1-2 битов на пару «ключ-значение».

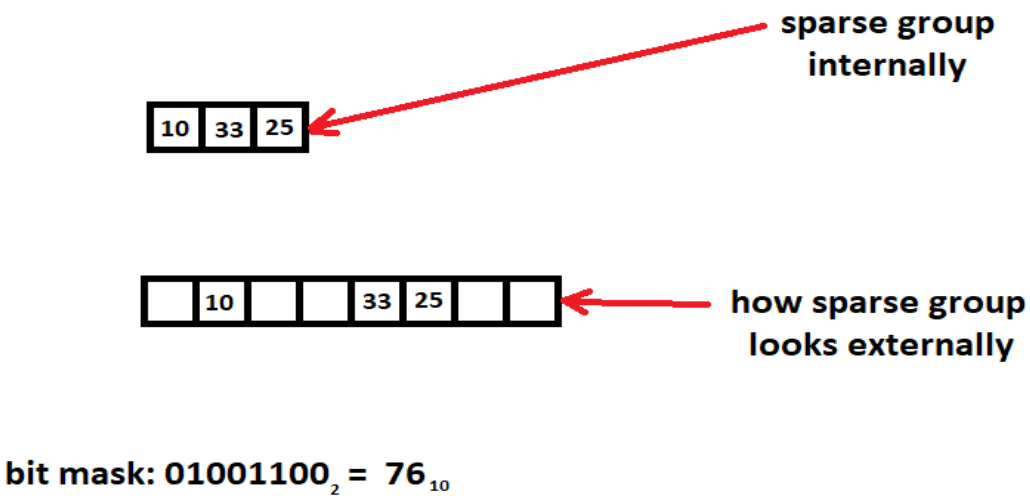
sparsetable - это простой динамический массив [3] из указателей на **sparse groups**.

2.1.1 Sparse group

Sparse group содержит в себе динамический массив, который реаллоцируется на **каждой** операции вставки/удаления (вместо реаллокаций при достижении размера степени двойки, как делает обычный динамический массив) максимального размера M как хранилище (в любой момент времени его размер $\leq M$, но для стороннего наблюдателя он ведёт себя так, как будто его размер ровно M) и битовую маску из M битов с информацией о том, какие места (видимые сторонним наблюдателем) на самом деле заполнены.
Начиная с этого момента, говоря «индекс» или «позиция», мы будем иметь в виду индекс, видимый сторонним наблюдателем.

Чтобы превратить внешний индекс во внутренний, надо сделать следующее: если даны внешний индекс $0 \leq N < M$ и битовая маска B размера M, мы должны найти количество единиц в B на местах $0...N-1$. Это может быть сделано любым способом (поскольку M - константа, даже простой цикл даст асимптотику времени $O(M) = O(1)$ в худшем случае), но реализация от Google использует несколько низкоуровневых оптимизаций для этого (использование массива из однобайтовых чисел как битовой маски вместо одного 4- или 8-байтового; простая таблица поиска из 256 элементов для преобразования внешнего индекса во внутренний, последовательно применённая к каждому числу из маски). То же верно для обратного преобразования.

Визуализация структуры sparse group:



Sparse group поддерживает 3 операции: **insert**, **delete**, и **random access**.

- **random access**: если дан индекс n, за $O(1)$ в худшем случае мы можем обратиться (записать или прочитать) элемент индекса n. Асимптотика этой операции достигается тривиально, т.к. хранилище - массив.

- **insert**: если дан ключ K и позиция n, вставить K на позицию n. Для этого нам придётся реаллоцировать массив, чтобы получить память на ещё один элемент. Это требует $(array\_size+1) \times memory\_for\_one\_element$ времени. Поскольку максимальный размер этого динамического массива равен M и мы выполняем вставку, $array\_size < M \implies array\_size + 1 \leq M$, а $memory\_for\_one\_element$ - константа, поэтому эта часть операции выполняется за $O(1)$ в худшем случае.
  Мы должны реаллоцировать память, потом скопировать элементы в соответствующие места, и после этого записать новый элемент в ячейку. Реаллокация требует $O(1)$ времени в худшем случае, копирование делается асимптотически за $O(1)$, т.к. размер копируемого блока ограничен сверху константой M, а запись нового элемента в ячейку - тоже $O(1)$ в худшем случае (из-за поддержки **random access**), поэтому **insert** выполняется за $O(1)$ времени в худшем случае (хотя константа довольно велика).

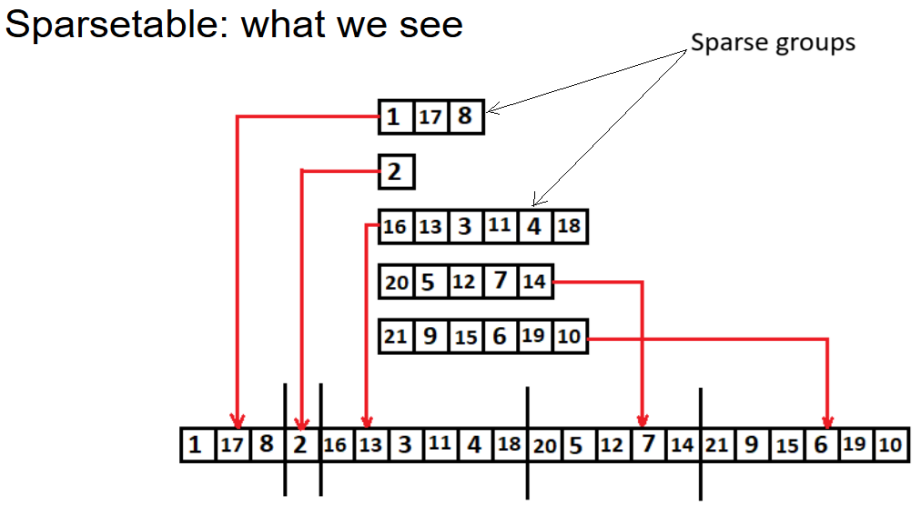
- **delete**: аналогично **insert**, требует $O(1)$ в худшем случае.

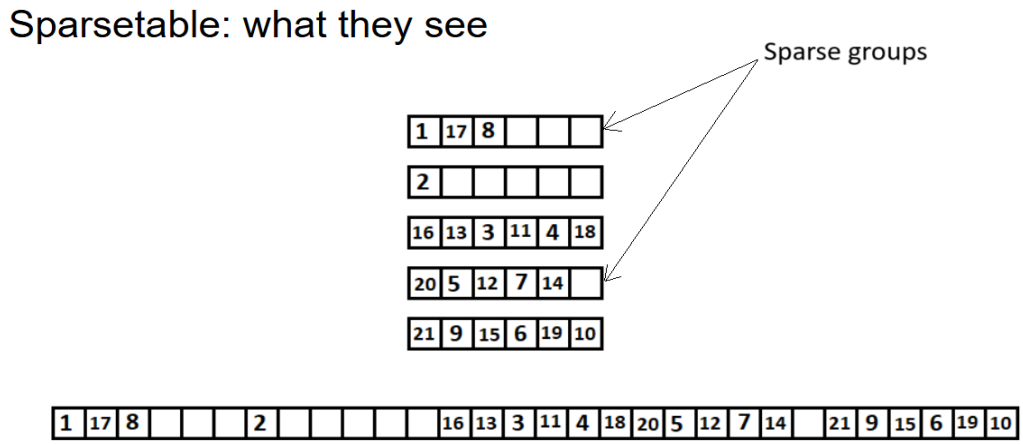
2.2 Внутреннее устройство sparsetable

Sparsetable представляет из себя динамический массив указателей на **sparse groups**. Обращение к элементу по индексу в **sparsetable** происходит следующим образом: вычисляется индекс **sparse group**, содержащей элемент по требуемому индексу,

затем индекс требуемого элемента внутри этой **sparse group**, и после этого непосредственно обращение к этому элементу (пусть мы хотим обратиться к элементу с индексом n, а максимально допустимый размер одной **sparse group** равен M, то, поделив n на M с остатком, получим равенство $n = M \times a + r$. Тогда требуемый элемент лежит в **sparse group** индекса a, а внутри неё он имеет индекс r).

Визуализация sparsetable (на верхнем рисунке внутреннее устройство, на нижнем - как это выглядит для стороннего наблюдателя):





2.2.1 Расход памяти

Использование лишней (т.е. потраченной не на непосредственно хранение пар «ключ-значение») памяти может проистекать из двух мест:

1. Пустые ячейки в динамическом массиве sparse group
2. Лишняя память внутри непустых sparse group

Если мы имеем N непустых sparse groups, у нас может быть только $L \leq N$ пустых (т.к. они содержатся в стандартном динамическом массиве). Каждая пустая sparse group содержит пустой массив, занимающий 0 памяти, и битовую маску из $M = O(1)$ битов, поэтому часть 1 занимает не более чем $L \times O(1) = O(L) \leq O(N)$ дополнительной памяти.
Непустая sparse group содержит массив и битовую маску. Массив использует всю свою память для хранения пар «ключ-значение» (т.к. в нём нет пустых ячеек), поэтому вся использованная дополнительная память используется битовыми масками. Аналогично предыдущему, они занимают $O(N)$ дополнительной памяти.

Константа M может быть выбрана пользователем, но авторы (Google) рекомендуют $M = 48$. Вычисления [4] показывают, что для такого выбора M использование дополнительной памяти лежит в пределах 1-2 бит на пару «ключ-значение».

2.3 Внутреннее устройство sparse-hash-table

Sparse-hash-table использует квадратичное пробирование [5] с открытой адресацией [6]. В качестве хранилища данных выступает sparsetable.

Открытая адресация - это метод разрешения хеш-коллизий, использующий один контейнер для хранения пар «ключ-значение». Чтобы добавить в неё пару «ключ-значение», мы должны вычислить хеш-функцию от ключа и с её помощью определить индекс ячейки, в которую мы попробуем вставить эту пару. При коллизии мы попробуем другую ячейку, потом третью и т.д. Правило, по которому определяется последовательность ячеек, называется пробированием.

Квадратичное пробирование работает просто: при коллизии на первой попытке вставки попробовать ячейку на 1 правее текущей. Если она занята - то сдвинуться на 2 ячейки вправо, затем на 3 и т.д. (считая, что хранилище «зацикливается» - т.е., отождествляя конец с началом). Оно называется квадратичным, потому что n-я ячейка, которую мы проверим, находится на расстоянии ($\frac{n(n-1)}{2}$ mod container_size) вправо от изначальной.

Несмотря на свою кажущуюся простоту и возраст (проблема была впервые сформулирована в 1968), до сих пор неизвестно, до какого load factor операции при квадратичном пробировании выполняются за $O(1)$ амортизированного времени. Лишь недавно было обнаружено [7], что load factor, гарантирующие $O(1)$ асимптотики, в принципе существуют (т.е. существует некоторое число $0 < \overline{\alpha} \leq 1$ такое, что любой load factor $\alpha < \overline{\alpha}$ даёт амортизированное $O(1)$ в асимптотиках).

Несмотря на это, на практике квадратичное пробирование работает хорошо. С этого момента мы будем предполагать, что все операции квадратичного пробирования работают за $O(1)$ амортизированного времени (т.е. мы предполагаем, что у нас есть некоторая хеш-функция, работающая за $O(1)$ в худшем случае, и что квадратичное пробирование может, принимая на вход ключ, вернуть индекс ячейки, содержащей этот ключ, или индекс ячейки, в которую мы можем вставить этот ключ, за $O(1)$ амортизированного времени для некоторых load factor'ов $0 \leq \alpha < c$).

Как и любая хеш-таблица, sparse-hash-table поддерживает 3 выполняющихся за $O(1)$ амортизированного времени (или лучше) операции: **insert**, **delete** и **lookup**.

- **lookup**: взять хеш-функцию ключа ($O(1)$ в худшем случае), затем осуществить lookup, смотря на sparsetable как на простой массив (количество сравнений квадратичного пробирования - $O(1)$ в среднем. Каждое сравнение выполняется за $O(1)$ в худшем случае элементарных операций для определения sparse group, содержащей данную ячейку (это выполняется простым делением хеш-функции от ключа на M с остатком), 0 дополнительных операций для вычисления индекса в этой sparse group (он равен остатку), $O(1)$ в худшем случае операций для преобразования внешнего индекса во внутренний, и $O(1)$ в худшем случае операций для обращения к этому элементу), поэтому lookup выполняется за $O(1)$ амортизированного времени.

  Псевдокод:

```
ind_in_sparsetable = hash(key_to_look_for)
step = 1
while step < sparsetable.max_size:
    sparsegroup_ind = ind_in_sparsetable // max_sparsegroup_size
    ind_in_sparsegroup = ind_in_sparsetable % max_sparsegroup_size
    current_group = sparsetable[sparsegroup_ind]
    current_cell = current_group[ind_in_sparsegroup]
    if current_cell.key == key_to_look_for:
        # match
        return current_cell.value
    else:
        # mismatch
        key = (key + step) % sparsetable.max_size
        step += 1
```

- **insert**: аналогично **lookup**, но в конце нам потребуется одно обращение на запись по известному внутреннему индексу известной sparse group (чтобы непосредственно записать пару «ключ-значение»). Эти действия выполняются за $O(1)$ в худшем случае, поэтому весь **insert** выполняется $O(1)$ амортизированного времени.

Псевдокод:

```
ind_in_sparsetable = hash(key_to_insert)
step = 1
while step < sparsetable.max_size:
    sparsegroup_ind = ind_in_sparsetable // max_sparsegroup_size
    ind_in_sparsegroup = ind_in_sparsetable % max_sparsegroup_size
    current_group = sparsetable[sparsegroup_ind]
    current_cell = current_group[ind_in_sparsegroup]
    if current_cell.key == sparsetable.empty_key or current_cell.key == sparsetable.tombstone_key:
        current_cell.key = key_to_insert
        current_cell.value = value.to_insert
        return
    else:
        key = (key + step) % sparsetable.max_size
        step += 1
```

- **remove**: аналогично **insert**, но последнее обращение заменяет элемент на **tombstone** (удалённый элемент). Требует асимптотически столько же времени, сколько **insert**.

  Псевдокод:

```
ind_in_sparsetable = hash(key_to_remove)
step = 1
while step < sparsetable.max_size:
    sparsegroup_ind = ind_in_sparsetable // max_sparsegroup_size
    ind_in_sparsegroup = ind_in_sparsetable % max_sparsegroup_size
    current_group = sparsetable[sparsegroup_ind]
    current_cell = current_group[ind_in_sparsegroup]
    if current_cell.key == key_to_remove:
        current_cell.key = sparsetable.tombstone_key
        current_cell.value = sparsetable.default_value
        return
    else:
        key = (key + step) % sparsetable.max_size
        step += 1
```

Также, мы обязаны иметь возможность сделать **resize** (удвоить размер) sparsetable за $O(n)$ в худшем случае, где n - количество элементов. Но это тривиально - стандартный динамический массив делает именно это, и ранее мы показали, что дополнительная память на один элемент равна в худшем случае $O(1)$ $\implies$ дополнительная память на всю таблицу равна в худшем случае $O(n)$.

3 Dense-hash-table

Dense-hash-table - таблица с открытой адресацией и квадратичным пробированием (см. раздел 2), использующая в качестве контейнера обычный массив.

Операции lookup, delete и insert для простого массива не влекут за собой дополнительного использования времени (т.е. всё используемое ими время заложено в квадратичное пробирование), поэтому все три операции выполняются за $O(1)$ амортизированного времени (разумеется, в предположении, что квадратичное пробирование работает за $O(1)$ амортизированного времени).
Время на resize - тоже $O(n)$ в худшем случае (следует из доказательства этого для простого динамического массива).

4 Сравнение

Как мы можем заметить, асимптотически sparse- и dense-hash-table эквивалентны. Впрочем, это верно для **любой** пары (корректно реализованных) хеш-таблиц, и мы на самом деле хотим сравнивать константы (по времени на каждую операцию, а также по затратам памяти на пару «ключ-значение»).

4.1 Память

Основная идея sparse-hash-table - оптимизировать расходы памяти настолько, насколько возможно, при этом оставаясь хеш-таблицей. Как показано в разделе 2, sparse-hash-table имеет лишь 1-2 бита дополнительной памяти на пару «ключ-значение», что очень мало.

Dense-hash-table, с другой стороны, имеет затраты памяти, зависящие от load factor. Авторы предлагают зафиксировать максимальный load factor на уровне 0.5 [8], поэтому мы будем использовать именно такой максимальный load factor для вычислений. Итак, load factor ≤ 0.5, поэтому количество пустых ячеек не меньше количества заполненных. Значит, если массив имеет размер m, у нас хотя бы $\frac{m}{2}$ пустых ячеек и не более чем $\frac{m}{2}$ заполненных, поэтому наши затраты дополнительной памяти равны $number\_of\_free\_spaces \times size\_of\_one\_entry \geq \frac{m}{2} \times size\_of\_one\_entry$. Разделив на $number\_of\_filled\_spaces \leq \frac{m}{2}$, мы получаем, что дополнительная память на одну использованную ячейку не меньше $size\_of\_one\_entry$ — то есть, у этой хеш-таблицы использование дополнительной памяти крайне велико: используемый размер не больше неиспользуемого.

4.2 Время

Часть использованного времени проистекает из квадратичного пробирования, которое есть в обеих таблицах, поэтому мы исключим эту часть времени из сравнения.

Как показано в разделе 3, dense-hash-table имеет **нулевое** дополнительное время поверх квадратичного пробирования. Sparse-hash-table, напротив, для lookup имеет такое же количество дорогих операций обращений к памяти (также, в ней вполне вероятны cache misses, т.к. sparse groups, наиболее вероятно, не будут расположены

рядом друг с другом), а также некоторые почти бесплатные из-за низкоуровневых оптимизаций вычисления для перевода внешнего индекса во внутренний (см. 2.1.1, параграф 2). При этом sparse groups имеют **большие** дополнительные затраты времени на реаллокацию памяти и копирование на каждый insert (если считать, что элементы распределены равномерно, то при уровне загрузки α мы будем должны сделать в среднем $\alpha \times M$ копирований. При рекомендованном $M = 48$ и желаемой нагрузкой $\alpha \geq 0.75$ получаем 36 копирований на каждую вставку), а также dense-hash-table по умолчанию поддерживает очень низкий максимальный уровень загрузки ($\alpha = 0.5$), поэтому dense-hash-table на практике существенно быстрее sparse-hash-table.

5 Больше способов использования sparsetable

Можно заметить, что использование sparsetable вместо простого массива при открытой адресации может превратить **любую** хеш-таблицу, использующую открытую адресацию и не имеющую дополнительных затрат памяти помимо пустых ячеек в массиве, в очень эффективную по памяти. Это даёт возможность создать много новых эффективных по расходу памяти хеш-таблиц, одна из которых будет представлена в этом разделе.

5.1 Hopscotch hashing

Hopscotch hashing - это некоторый способ модифицировать линейное пробирование. Известно, что он может выдавать в среднем более хорошую скорость, поэтому мы хотим попробовать именно его для создания новой хеш-таблицы, которая была бы эффективна по памяти и при этом могла соперничать по скорости с sparse-hash-table.

Основная идея hopscotch hashing [9] заключается в использовании линейного пробирования, при этом иногда переставляя содержимое ячеек для того, чтобы максимальное расстояние между ячейкой, полученной в результате взятия хеш-функции от ключа, и ячейкой, в которой лежит пара «ключ-значение» с этим ключом, было не больше некоторой константы K (в тестах используется $K = 32$, а также верхнее ограничение на количество ячеек, которые мы проверяем при линейном пробировании, равное 128).

Подробнее про hopscotch hashing [10]

5.1.1 Варианты реализации

У hopscotch hashing есть разные варианты реализации. Описанная в [10] использует битовые маски на каждый bucket - т.е. по битовой маске размера K на каждую ячейку в хранилище, вне зависимости от того, заполнена она или нет. Это использует существенное количество дополнительной памяти, поэтому такой способ нам не подходит.
Однако, в статье 2013 года [11] Emmanuel Goossaert пишет о так называемой shadow representation - «теневой» интерпретации hopscotch hashing. Её основная идея заключается в следующем: при использовании hopscotch hashing наша главная проблема - быстро определять, к какому bucket принадлежит пара «ключ-значение» из данной ячейки (в методе разрешения этой проблемы и расходятся разные интерпретации

hopscotch hashing).
Интерпретация с битовыми масками вообще не имеет данной проблемы [10].
«Теневая» интерпретация решает её так: имея пару «ключ-значение», мы можем просто ещё раз вычислить хеш-функцию от ключа и таким образом определить bucket. Это может занимать много времени при использовании «дорогих» хеш-функций - поэтому мы в практических измерениях будем использовать «дешевые» (больше об этом в следующем разделе).

5.1.2 Теневая интерпретация

Большое преимущество теневой интерпретации в том, что она не требует дополнительной памяти. Поэтому мы можем сделать таблицу, использующую открытую адресацию, теневую интерпретацию hopscotch hashing в качестве пробирования и sparsetable в качестве хранилища.

5.1.3 Асимптотические оценки

В [10] доказаны асимптотически оценки для интерпретации с битовыми масками.
Здесь мы приведём асимптотические оценки для теневой интерпретации.

- **insert**: здесь дополнительное время проистекает только из взятия хеш-функции при каждой перестановке двух элементов друг с другом. Известно [10], что таких перестановок - амортизированно $O(1)$, поэтому в теневой интерпретации **insert** также выполняется за $O(1)$ амортизированного времени.

- **remove**: отличий от интерпретации с битовыми масками здесь два, и оба проистекают из того, что теневая интерпретация использует tombstone вместо удаления. Во-первых, мы не удаляем элемент, а заменяем - впрочем, здесь нам нужно сделать даже меньше действий, потому что нам не нужно обновлять соответствующую битовую маску. Во-вторых, само нахождение ячейки с данным ключом теоретически может занимать большее время из-за наличия tombstone в таблице, но мы будем считать, что количество tombstone не превышает количества реальных элементов (при превышении - будем реаллоцировать таблицу в новую таблицу **того же** размера, копируя элементы по одному, не учитывая tombstone. Очевидно, что эта операция может быть вызвана на таблицу с n реальными элементами только после хотя бы $n+1$ удалений, при этом эта операция выполняется за $O(n)$ в худшем случае - поэтому асимптотики остаются теми же), значит, количество занятых ячеек не более чем в 2 раза больше, чем количество реально существующих элементов - а это константа. Поэтому при удвоении количества занятых ячеек в оценках из [10] амортизированное $O(1)$ на эти операции превращается в $O(2) = O(1)$, то есть асимптотика сохранена.

- **contains**: доказано выше.

Таким образом, теневая интерпретация hopscotch hashing действительно является хеш-таблицей, и поэтому описанная в начале 5.1.2 хеш-таблица (использующая sparsetable как хранилище) действительно корректна и (если не учитывать, на каких load factor обычно происходит resize у hopscotch hashing и у квадратичного пробирования) так же эффективна по памяти, как и sparse-hash-table (т.к. она тоже использует

tombstone).

Важно заметить, что варианты реализации (интерпретации) hopscotch hashing не влияют на положение элементов внутри таблицы и частоту роста максимального размера, а влияют только на время и требуемую память работы операций. Поэтому все оценки на максимальный уровень загрузки, при которых эта хеш-таблица действительно имеет асимптотику $O(1)$ на поиск, верные для интерпретации с битовыми масками (доказанные в [10]), также верны и здесь (если учитывать tombstone как существующие элементы при расчёте уровня загрузки. Если же их не учитывать - то все оценки делятся на 2, т.к. tombstone ведут себя как существующие элементы для поиска, и их (см. выше) будет не больше, чем существующих элементов).

6 Практические измерения

6.1 Условия экспериментов

Были проведены измерения среднего времени операций (insert, remove, contains с ответом «да», contains с ответом «нет») на количествах элементов $10^3, 10^4, ..., 10^7$ (в случае insert измерялась длина пути от 0 элементов до требуемого количества, в случае remove - длина пути от данного количества до 0 элементов). Все измерения проводились на C++ с флагом оптимизации -O3, компилятором GCC 13.3.0. Для sparse- и dense-hash-table используются реализации от Google [2], для интерпретации hopscotch с битовыми масками - реализация из моей курсовой работы прошлого года [10], теневая интерпретация hopscotch использует sparsetable от Google [2] и мою реализацию hopscotch. Весь исходный код, включая тесты и бенчмарки, находится в общем доступе на GitHub [12].
Все таблицы в качестве хеш-функции используют std::hash, в качестве ключа int. Для простоты использовалось пустое значение (в частности, среди вариаций таблиц от Google использовались sparse-hash-set и dense-hash-set, а не -table).

Более подробно:

- **insert**: имея пустую хеш-таблицу, измеряем время на вставку N различных элементов
- **remove**: имея хеш-таблицу с N элементами, измеряем время на удаление всех её элементов (в случайном порядке)
- **true contains**: имея хеш-таблицу с N элементами, измеряем суммарное время на вызов **contains** для каждого из элементов (в случайном порядке)
- **false contains**: имея хеш-таблицу с N элементами и (динамический) массив из N чисел, которых гарантированно нет в этой таблице, измеряем суммарное время на вызов **contains** для каждого элемента этого массива

6.2 Результаты

Результаты тестирования представлены на рисунках 1-4 (на каждом из них по оси X идёт количество элементов, по оси Y - среднее время в наносекундах на одну операцию соответствующего типа (insert, remove, true contains, false contains)).

Рис. 1:

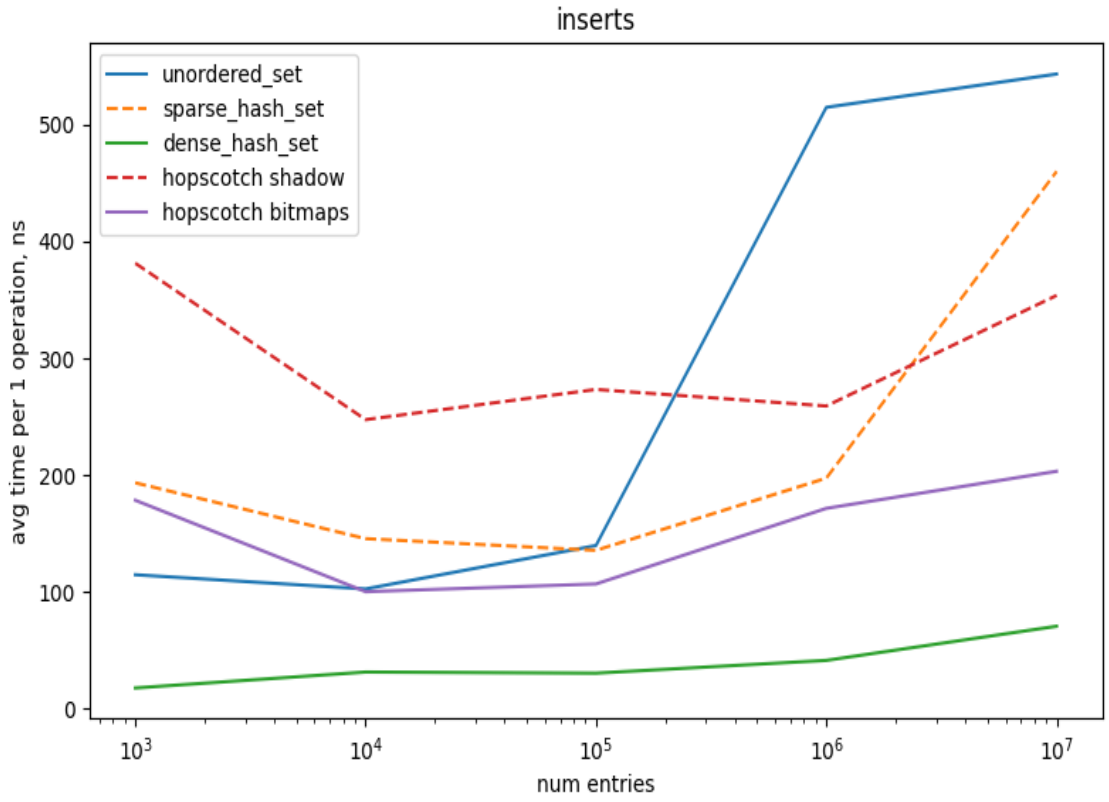

inserts
unordered_set
sparse_hash_set
dense_hash_set
hopscotch shadow
hopscotch bitmaps
avg time per 1 operation, ns
0
100
200
300
400
500
10^3
10^4
10^5
10^6
10^7
num entries

Рис. 2:

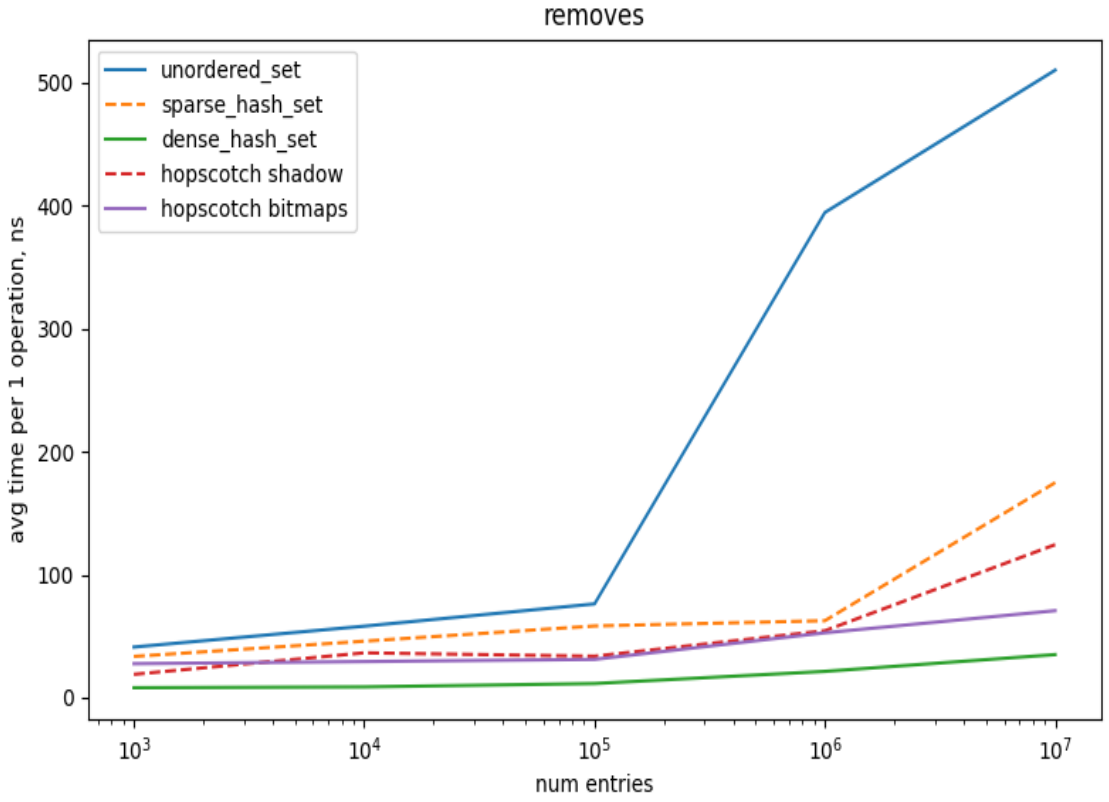

removes
unordered_set
sparse_hash_set
dense_hash_set
hopscotch shadow
hopscotch bitmaps
avg time per 1 operation, ns
0
100
200
300
400
500
10^3
10^4
10^5
10^6
10^7
num entries

Рис. 3:

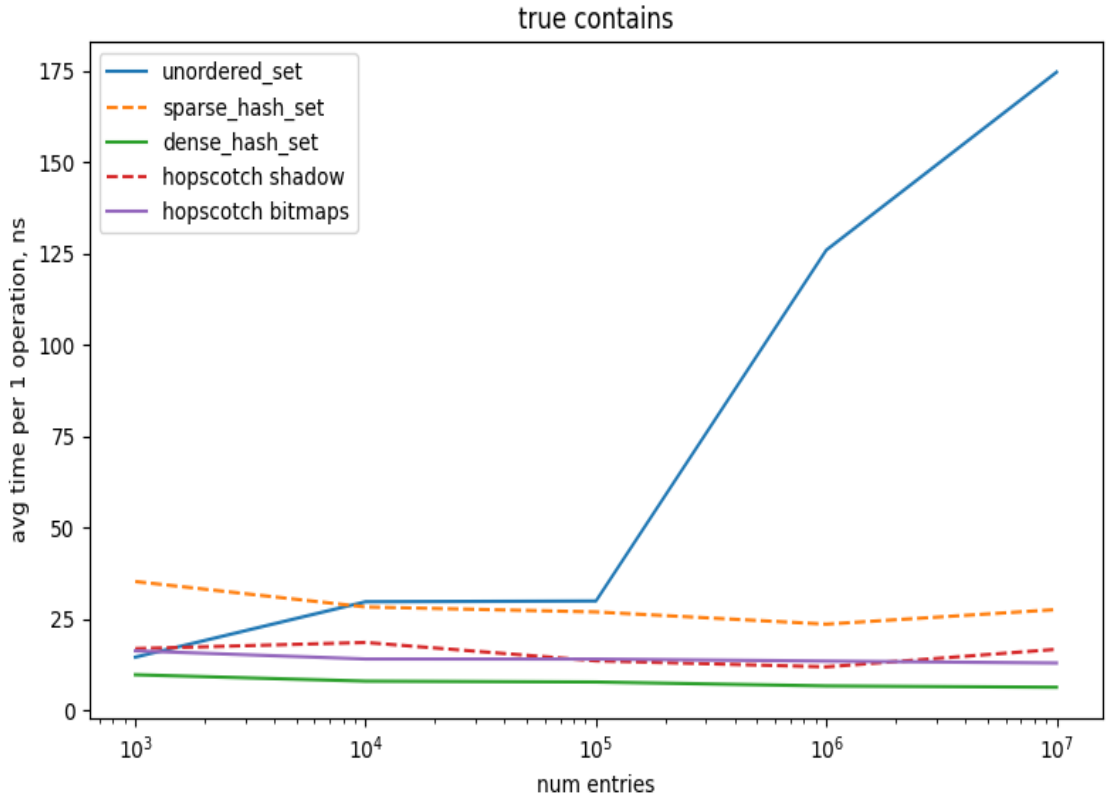
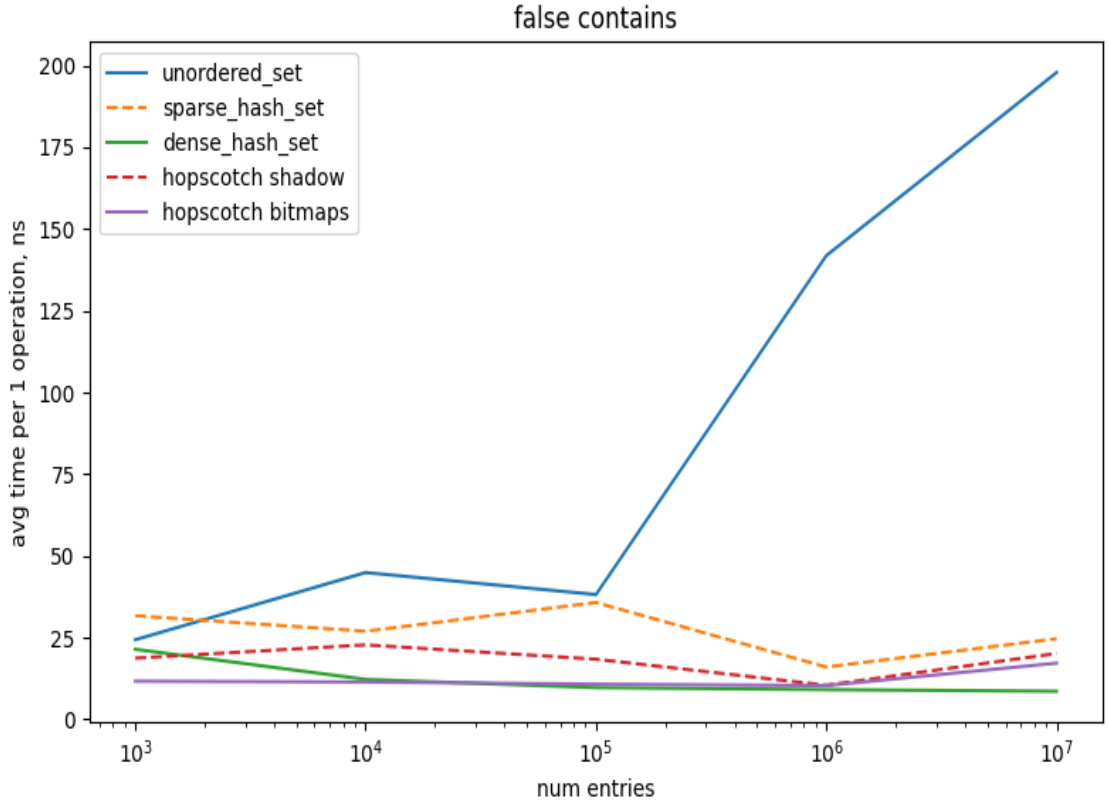

true contains
unordered_set
sparse_hash_set
dense_hash_set
hopscotch shadow
hopscotch bitmaps
avg time per 1 operation, ns
num entries
0
25
50
75
100
125
150
175
10^3
10^4
10^5
10^6
10^7

Рис. 4:


false contains
unordered_set
sparse_hash_set
dense_hash_set
hopscotch shadow
hopscotch bitmaps
avg time per 1 operation, ns
num entries
0
25
50
75
100
125
150
175
200
10^3
10^4
10^5
10^6
10^7

Как видно из тестов, теневая интерпретация hopscotch hashing (красная пунктирная линия) обгоняет sparse-hash-set (оранжевая пунктирная линия) на всех операциях, кроме insert, причём на true contains результат значителен (в среднем примерно в 1.8 раз), сохраняя ту же эффективность по памяти.

Список литературы

[1] Wikipedia contributors. (2025, May 24). Hash table. In *Wikipedia, The Free Encyclopedia*. Retrieved May 30, 2025, from https://en.wikipedia.org/w/index.php?title=Hash_table&oldid=1291994466

[2] Google. (2020, August 12). Sparsehash. Retrieved May 30, 2025, from https://github.com/sparsehash/sparsehash/tree/master

[3] Wikipedia contributors. (2025, May 26). Dynamic array. In *Wikipedia, The Free Encyclopedia*. Retrieved May 30, 2025, from https://en.wikipedia.org/w/index.php?title=Dynamic_array&oldid=1292303927

[4] Google. (2016, July 26). Sparsehash. Retrieved May 30, 2025, from https://github.com/sparsehash/sparsehash/blob/master/doc/implementation.html.
Раздел «Resource use» у sparsetable

[5] Wikipedia contributors. (2024, November 26). Quadratic probing. In *Wikipedia, The Free Encyclopedia*. Retrieved May 30, 2025, from https://en.wikipedia.org/w/index.php?title=Quadratic_probing&oldid=1259630039

[6] Wikipedia contributors. (2025, March 1). Open addressing. In *Wikipedia, The Free Encyclopedia*. Retrieved May 30, 2025, from https://en.wikipedia.org/w/index.php?title=Open_addressing&oldid=1278336144

[7] William Kuszmaul, Zoe Xi, *Towards an Analysis of Quadratic Probing (2024)*

[8] Google. (2016, July 12). Sparsehash. Retrieved May 30, 2025, from https://github.com/sparsehash/sparsehash/blob/master/src/sparsehash/internal/densehashtable.h#L1311

[9] Wikipedia contributors. (2024, December 18). Hopscotch hashing. In *Wikipedia, The Free Encyclopedia*. Retrieved May 30, 2025, from https://en.wikipedia.org/w/index.php?title=Hopscotch_hashing&oldid=1263768861

[10] Гущин А.И., *Методы разрешения хеш-коллизий. Hopscotch hashing (2024)*

[11] Goossaert, E. (2013, August 11). Hopscotch hashing. Retrieved April 10, 2025, from https://codecapsule.com/2013/08/11/hopscotch-hashing/

[12] Guschin, A.I. (2025, May 10). Optimized Hash Collision Resolution Methods. Source code. Retrieved May 10, 2025, from https://github.com/aiguschin/hashtables-and-benches