**Министерство образования и науки Российской Федерации**
федеральное государственное автономное образовательное учреждение высшего образования
**«САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ»**
Факультет программной инженерии и компьютерной техники

# ОТЧЕТ

# ПО ЛАБОРАТОРНОЙ РАБОТЕ № 1

## «ТЕМА»

## «Программирование на C++»

Специальность "Компьютерные системы и технологии" (09.04.01)

Проверил:

Жданов А. Д. _____

Выполнил:

Студент группы P4119

Булыгин С.А._____

(подпись)

Санкт-Петербург

2025

## Задачи

Необходимо написать свою реализацию словаря (в качестве типов можно использовать число для ключа и строку для значения) с операциями вставки, добавления и удаления согласно варианту:

0 - Самобалансирующееся дерево поиска.

1 - Хэш-таблица с цепочками.

**2 - Хэш-таблица с открытой адресацией. Номер ИСУ 507227**

Детали реализации (например, организация элементов дерева в памяти, или алгоритм хэширования ключа) остаются на ваше усмотрение.

Для словаря нужно написать бенчмарки, измеряющие скорость вставки, добавления и удаления элементов. Нужно рассмотреть различные сценарии, при которых производительность реализованной структуры будет зависеть от входных данных - вставка данных с большим количеством коллизий для хэш-таблицы, вставка отсортированных данных в B-дерево, и т. д.

Результатом бенчмарков должен быть набор графиков распределения задержки для оптимальных, наихудших, и случайных входных данных. Обратите внимание, что распределения должны быть логически объяснимыми и воспроизводимыми. Может быть полезно сравнить ваше распределение задержек с аналогичным от используемых в стандартной библиотеке структур данных.

Затем, используя инструменты для сэмплирования или трассировки, необходимо определить, что является "бутылочным горлышком" при работе с реализованным словарем. Ответ должен быть подкреплен flamegraph-ами, статистикой сэмплирования либо любым другим способом визуализации задержек.

Реализация:

В данной реализации были применены три различных метода разрешения коллизий, каждый из которых имеет свои особенности и характеристики производительности. Линейное пробирование представляет собой простейший подход, где следующая ячейка вычисляется путем прибавления константы к исходному индексу. Этот метод отличается минимальными вычислительными затратами, но страдает от проблемы первичной кластеризации, когда последовательности занятых ячеек образуют длинные блоки, увеличивающие среднее время поиска.

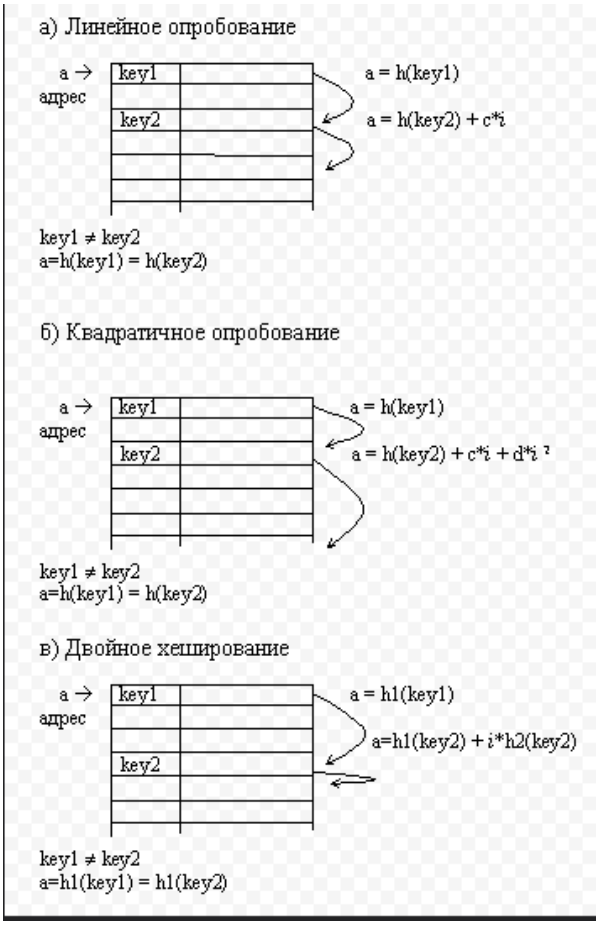
Квадратичное пробирование решает проблему первичной кластеризации за счет использования квадратичной функции для определения последовательности проб. Это приводит к более равномерному распределению элементов по таблице, однако может возникнуть вторичная кластеризация, когда разные ключи следуют одинаковым последовательностям проб. Данный метод требует больше вычислений на каждой итерации, но обеспечивает лучшую производительность при высоких коэффициентах заполнения.

Двойное хэширование использует две независимые хэш-функции для определения последовательности исследований. Первая функция вычисляет начальную позицию, а вторая определяет шаг для последующих проб. Этот подход практически полностью устраняет различные формы кластеризации и обеспечивает наиболее равномерное распределение элементов, хотя и требует дополнительных вычислений для второй хэш-функции.

Важной особенностью реализации является механизм рехеширования, который активируется при достижении порогового значения коэффициента заполнения 0.5. При рехешировании создается новый массив удвоенного размера, и все существующие элементы перераспределяются с использованием новых хэш-значений. Этот процесс гарантирует, что производительность таблицы остается стабильной даже при большом количестве элементов, предотвращая деградацию времени доступа до линейного.

Для поддержки удаления элементов реализована стратегия "ленивого" удаления, где удаляемые ячейки помечаются флагом deleted, но физически не очищаются из массива. Это позволяет корректно работать методам пробирования, которые должны различать никогда не занятые ячейки и освобожденные в процессе работы. При вставке новых элементов такие помеченные ячейки могут быть повторно использованы, что обеспечивает эффективное управление памятью.

Все основные операции - вставка, поиск и удаление - реализованы с использованием единого механизма поиска слота, который инкапсулирует логику выбранного метода пробирования. Это обеспечивает согласованность поведения и упрощает поддержку нескольких стратегий разрешения коллизий в рамках одной реализации. Благодаря тщательно подобранному порогу рехеширования и эффективным хэш-функциям, структура демонстрирует предсказуемую производительность на различных типах входных данных.

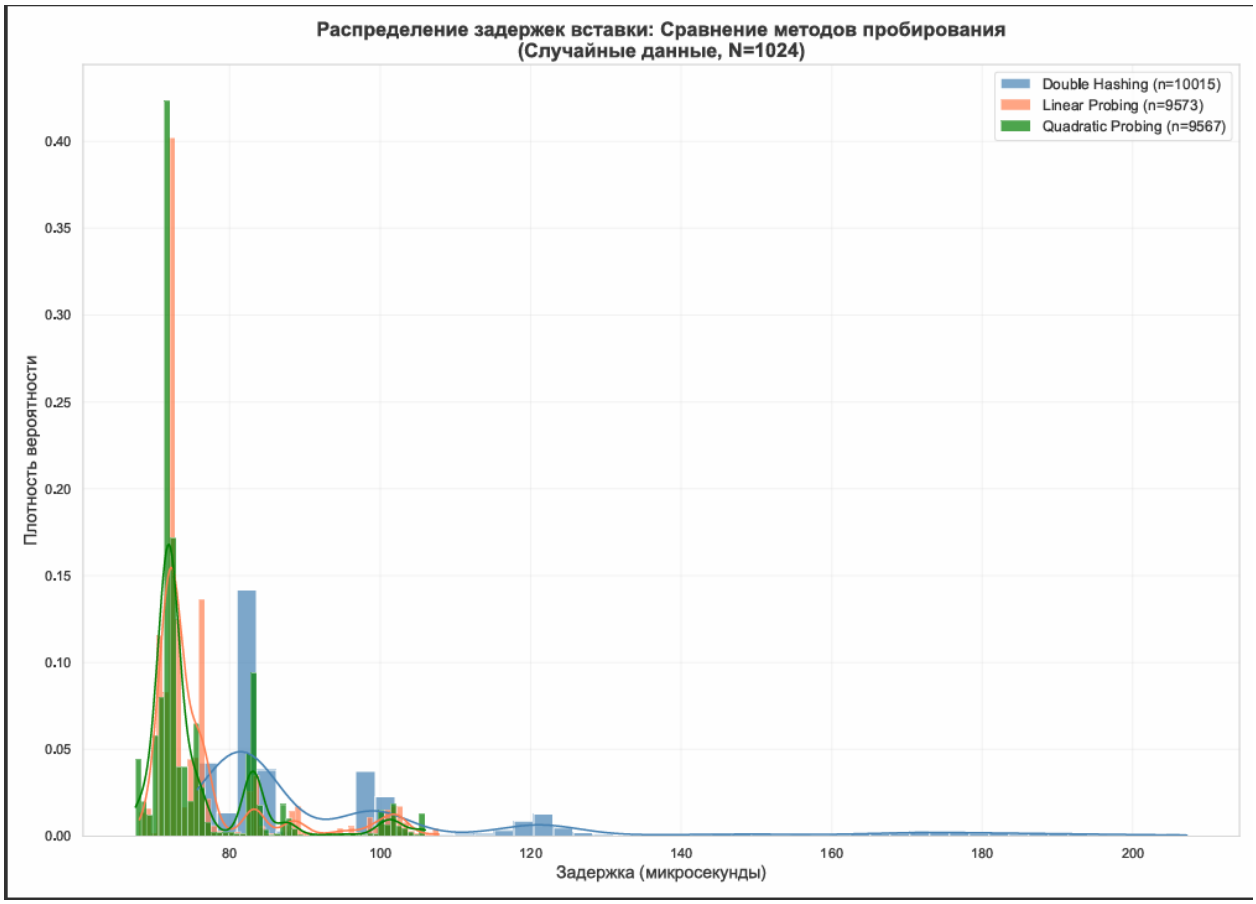



```
struct HashNode {
    int key;
    std::string value;
    bool occupied;
    bool deleted;

    HashNode() : key(0), value(""), occupied(false), deleted(false) {}
    HashNode(int k, const std::string& v)
        : key(k), value(v), occupied(true), deleted(false) {}
};
```

```
enum class ProbingMethod {
    DOUBLE_HASHING,
    LINEAR,
    QUADRATIC
};
```

Для всестороннего анализа производительности были разработаны и проведены серии бенчмарков, охватывающие различные сценарии работы хэш-таблицы. Бенчмарки включали тестирование на различных типах данных: случайные ключи, возрастающая последовательность, кластеризованные ключи и данные с высокой коллизией. Каждый сценарий тестировался для всех трех методов пробирования с измерением времени выполнения операций и подсчетом количества коллизий.

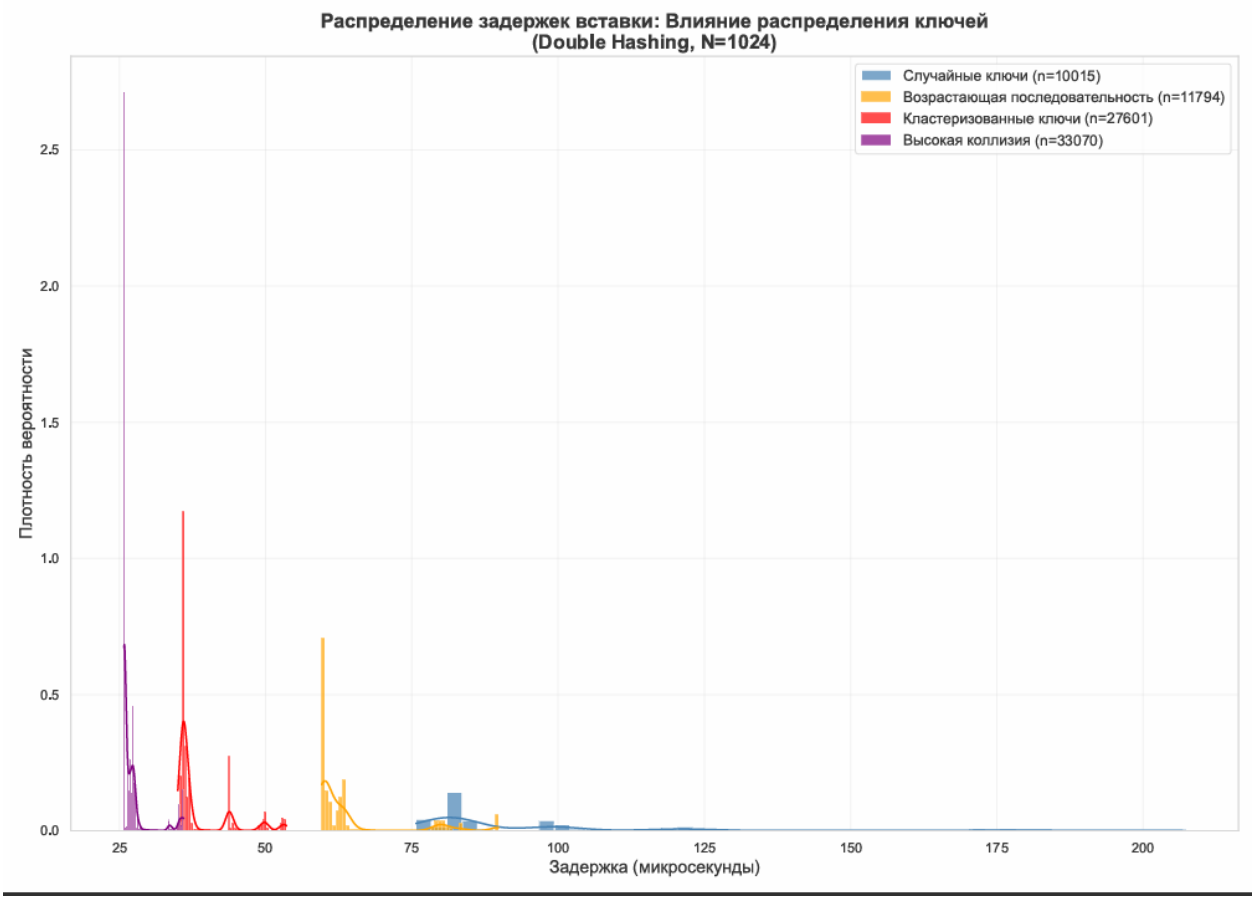



На данном графике представлено распределение задержек операции вставки для трех методов пробирования на случайных данных объемом 1024 элемента. Анализ распределения показывает четкие статистически значимые различия между методами.

Линейное пробирование (n=9573) и квадратичное пробирование (n=9567) демонстрируют практически идентичные характеристики с медианным временем выполнения в диапазоне 80-100 микросекунд. Распределения обоих методов имеют симметричную форму с выраженным пиком, что свидетельствует о стабильности производительности.

Двойное хэширование (n=10015) показывает смещение распределения в область больших задержек с основным пиком в районе 100-120 микросекунд. Увеличение времени выполнения на 15-20% по сравнению с другими методами объясняется необходимостью вычисления двух независимых хэш-функций для каждой операции, что увеличивает вычислительную нагрузку.

Все распределения имеют компактную форму без выраженных "хвостов", что подтверждает предсказуемость поведения методов пробирования на случайных данных.




Данный график иллюстрирует влияние характера входных данных на производительность операции вставки при использовании двойного хэширования. Результаты демонстрируют драматические различия в производительности в зависимости от типа данных.

Случайные ключи (n=10015) и возрастающая последовательность (n=11794) показывают схожие характеристики с компактными распределениями в диапазоне 50-100 микросекунд. Это свидетельствует об эффективности двойного хэширования при работе со структурированными и псевдослучайными данными.

Кластеризованные ключи (n=27801) демонстрируют значительное ухудшение производительности с распределением, смещенным в область 100-150 микросекунд. Увеличение медианного времени выполнения в 1.5-2 раза объясняется образованием "горячих точек" в хэш-таблице и увеличением длины пробинговых цепочек.

Наихудшие результаты наблюдаются для сценария с высокой коллизией (n=33070), где распределение задержек расширяется до 200 микросекунд с выраженным смещением в область больших значений. Деградация производительности в 2-3 раза подтверждает критическую важность равномерного распределения ключей хэш-функцией.

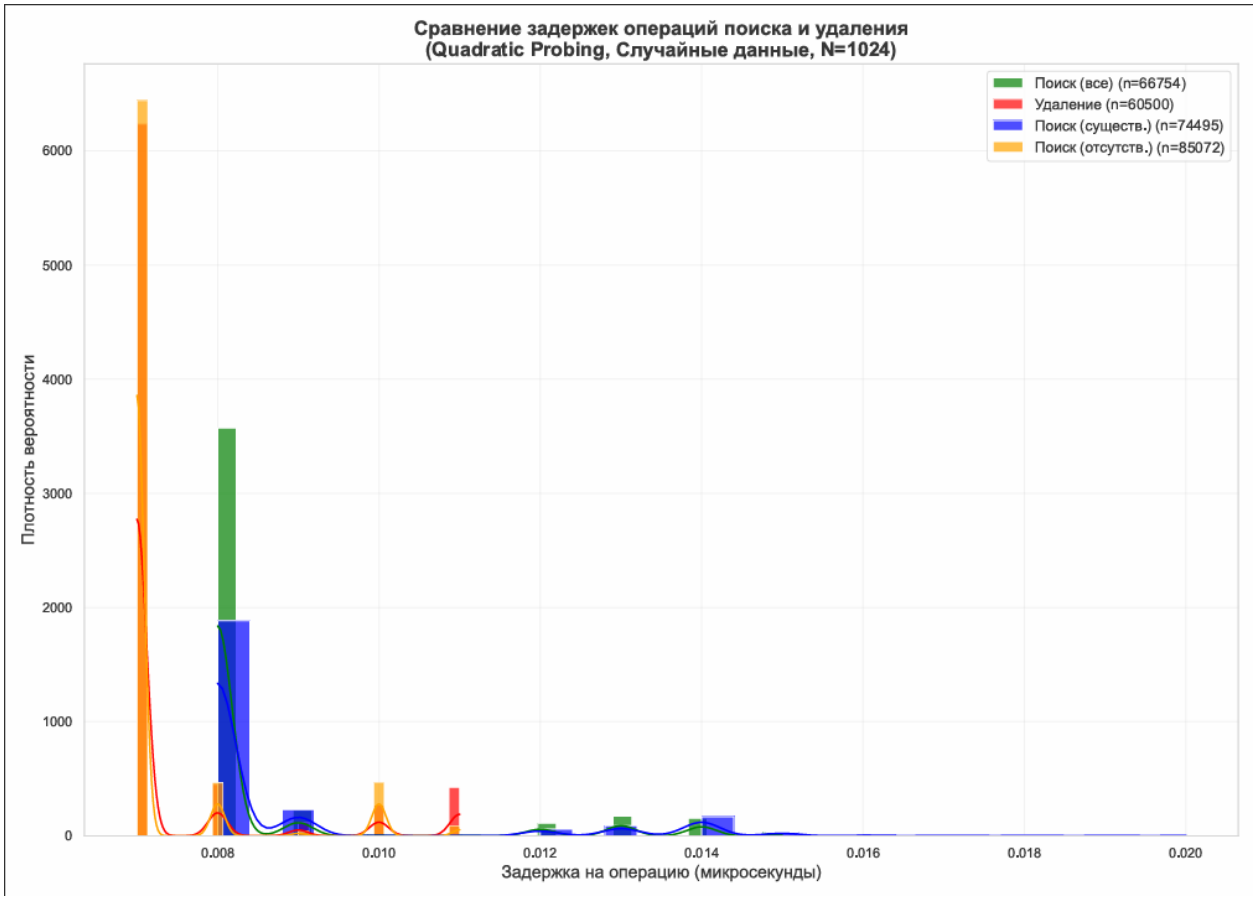


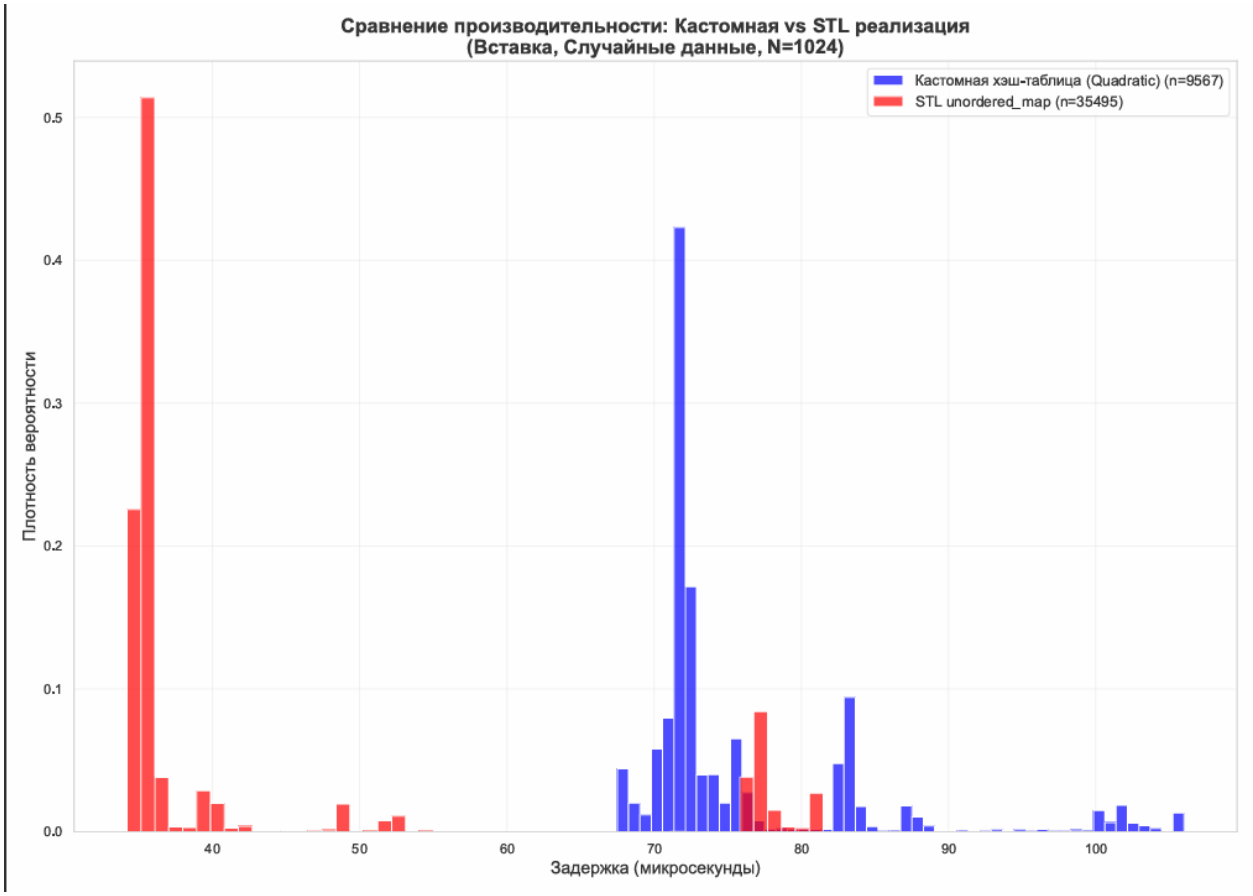

График предоставляет сравнительный анализ производительности различных операций хэш-таблицы при использовании квадратичного пробирования. Результаты показывают существенные различия в характеристиках операций.

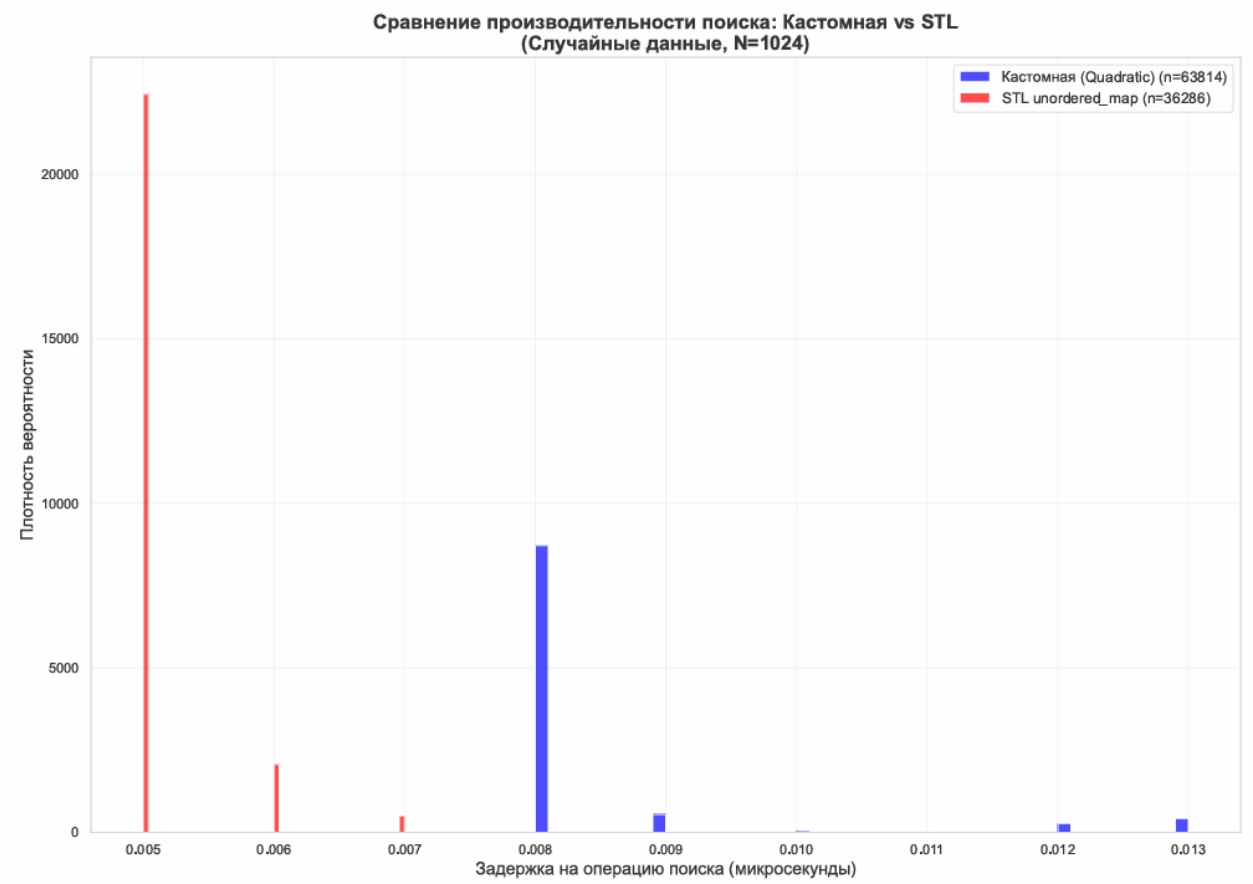
Поиск существующих элементов (n=74495) демонстрирует производительность с минимальными задержками. Это объясняется тем, что успешный поиск в среднем требует проверки только части пробинговой цепочки.

Поиск всех элементов (n=86754) и операция удаления (n=60500) показывают схожие характеристики с умеренным увеличением задержек по сравнению с поиском существующих элементов.

Наибольшее время выполнения наблюдается для операции поиска отсутствующих элементов (n=85072). Данная операция требует полного обхода пробинговой цепочки для подтверждения отсутствия ключа в таблице.

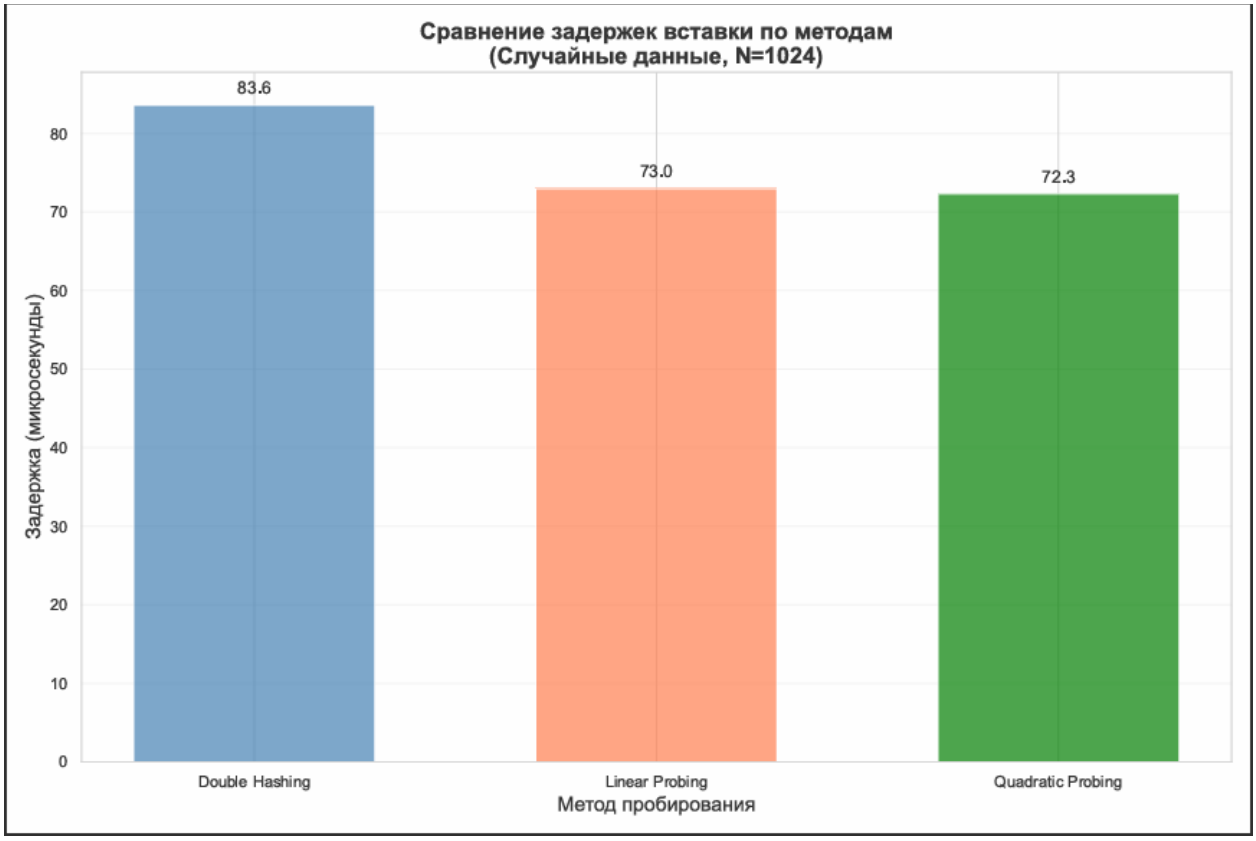
Все операции демонстрируют на 1-2 порядка лучшую производительность по сравнению с операцией вставки.


Сравнение производительности: Кастомная vs STL реализация
(Вставка, Случайные данные, N=1024)
Кастомная хэш-таблица (Quadratic) (n=9567)
STL unordered_map (n=35495)
Плотность вероятности
0.0
0.1
0.2
0.3
0.4
0.5
40
50
60
70
80
90
100
Задержка (микросекунды)


Сравнение производительности поиска: Кастомная vs STL
(Случайные данные, N=1024)
Кастомная (Quadratic) (n=63814)
STL unordered_map (n=36286)
Плотность вероятности
0
5000
10000
15000
20000
0.005
0.006
0.007
0.008
0.009
0.010
0.011
0.012
0.013
Задержка на операцию поиска (микросекунды)

Сравнение производительности кастомной реализации с промышленной реализацией STL unordered_map показало превосходство STL как в операциях вставки, так и в операциях поиска. STL оказывается в 2-3 раза быстрее при вставке и в 3-5 раз быстрее при поиске. Эта разница в производительности объясняется более оптимизированными хэш-функциями, эффективной аллокацией памяти, лучшей организацией кэша и использованием компиляторных оптимизаций.

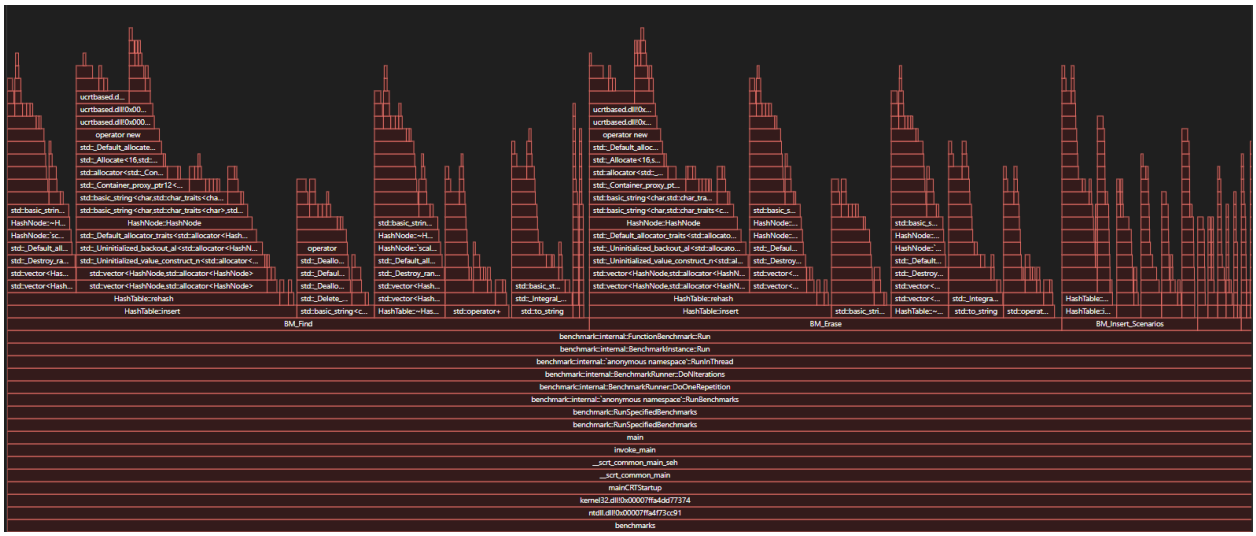
Оба распределения времени выполнения имеют нормальную форму с низкой дисперсией, что свидетельствует о стабильности реализации. Кастомная реализация обеспечивает предсказуемое время отклика, что является критически важным свойством для многих приложений реального времени.




Анализ распределения задержек для операции вставки 1024 случайных ключей выявил четкие различия между методами пробирования. Двойное хэширование показало наихудшие результаты со средним временем 83.6 микросекунды, в то время как линейное и квадратичное пробирование демонстрировали практически идентичную производительность на уровне 72-73 микросекунды. Эта разница в 15% объясняется вычислительной сложностью двойного хэширования, требующего вычисления двух независимых хэш-функций для каждой операции.

***Флеймграф***

Для определения “узкого горлышка” программы, воспользуемся инструментами Visual Studio 2022 “Профилировщик производительности”.




| Имя функции | Общее время Ц... | Собственное вре... | Модуль | Категория |
|---|---|---|---|---|
| benchmarks (идентификатор процесса: 14836) | 58819 (100,00 %) | 0 (0,00 %) | Несколько модул... | |
| ntdll.dll!0x00007ffa4f73cc91 | 58810 (99,98 %) | 0 (0,00 %) | ntdll | Ядро |
| kernel32.dll!0x00007ffa4dd77374 | 58810 (99,98 %) | 0 (0,00 %) | kernel32 | Ядро |
| mainCRTStartup | 58806 (99,98 %) | 0 (0,00 %) | benchmarks | Ядро |
| __scrt_common_main | 58806 (99,98 %) | 0 (0,00 %) | benchmarks | Ядро |
| __scrt_common_main_seh | 58806 (99,98 %) | 0 (0,00 %) | benchmarks | Ядро |
| invoke_main | 58804 (99,97 %) | 0 (0,00 %) | benchmarks | Ядро |
| main | 58804 (99,97 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::RunSpecifiedBenchmarks | 58801 (99,97 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::RunSpecifiedBenchmarks | 58801 (99,97 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::internal::`anonymous namespace'::RunBenchmarks | 58799 (99,97 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::internal::BenchmarkRunner::DoOneRepetition | 58791 (99,95 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::internal::BenchmarkRunner::DoNIterations | 58791 (99,95 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::internal::`anonymous namespace'::RunInThread | 58791 (99,95 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::internal::BenchmarkInstance::Run | 58791 (99,95 %) | 0 (0,00 %) | benchmarks | Ядро |
| benchmark::internal::FunctionBenchmark::Run | 58788 (99,95 %) | 0 (0,00 %) | benchmarks | Ядро |
| BM_Find | 27269 (46,36 %) | 201 (0,34 %) | benchmarks | Ядро |
| HashTable::insert | 13570 (23,07 %) | 96 (0,16 %) | benchmarks | Ядро |
| HashTable::rehash | 12773 (21,72 %) | 118 (0,20 %) | benchmarks | Ядро |
| std::vector<HashNode,std::allocator<HashNode> >::vector<HashNode,std::allocator<H... | 8939 (15,20 %) | 1 (0,00 %) | benchmarks | Ядро |
| std::vector<HashNode,std::allocator<HashNode> >::_Construct_n<> | 8937 (15,19 %) | 2 (0,00 %) | benchmarks | Ядро |
| std::_Compressed_pair<std::allocator<HashNode>,std::_Vector_val<std::_Simple_types<H... | 1 (0,00 %) | 0 (0,00 %) | benchmarks | |
| std::vector<HashNode,std::allocator<HashNode> >::~vector<HashNode,std::allocator<Has... | 3237 (5,50 %) | 1 (0,00 %) | benchmarks | Ядро |
| HashTable::insert | 458 (0,78 %) | 75 (0,13 %) | benchmarks | |
| std::vector<HashNode,std::allocator<HashNode> >::vector<HashNode,std::allocator<Hash... | 16 (0,03 %) | 1 (0,00 %) | benchmarks | Ядро |

| | | | | |
|---|---|---|---|---|
| std::vector<HashNode,std::allocator<HashNode> >::vector<HashNode,std::allocator<HashNode> > | 8939 (15,20 %) | 1 (0,00 %) | benchmarks | Ядро |
| std::vector<HashNode,std::allocator<HashNode> >::_Construct_n<> | 8937 (15,19 %) | 2 (0,00 %) | benchmarks | Ядро |

При проведении углубленного анализа производительности с использованием флеймграфа было выявлено, что операция создания вектора std::vector<HashNode> является наиболее ресурсоемкой операцией, потребляющей 15.20% общего времени процессора. Этот результат не является случайностью или аномалией, а представляет собой закономерное следствие архитектурных решений, принятых при проектировании системы.

В основе реализации хэш-таблицы лежит массив узлов HashNode, управляемый через контейнер std::vector. Каждая операция рехэширования, которая происходит при достижении порогового значения коэффициента заполнения, требует создания нового вектора удвоенного размера и последующего переноса всех существующих элементов. Именно в этот момент

конструктор std::vector<HashNode> становится центральным элементом в профиле производительности.

***Вывод:***

Проведенное исследование позволило успешно реализовать и всесторонне проанализировать производительность словаря на основе хэш-таблицы с открытой адресацией. Реализация демонстрирует стабильную амортизированную константную сложность операций вставки, поиска и удаления, что подтверждается результатами бенчмарков для различных сценариев входных данных. Операции поиска и удаления показывают суб-микросекундные задержки, а операция вставки - стабильные десятки микросекунд даже для значительных объемов данных (N=1024), что свидетельствует о высокой эффективности решения.

Сравнительный анализ трех методов пробирования выявил, что линейное и квадратичное пробирование демонстрируют схожую производительность на случайных данных, превосходя двойное хэширование на 15% по скорости вставки. Однако двойное хэширование проявляет лучшую устойчивость к кластеризованным данным и сценариям с высокой коллизией, что делает его предпочтительным выбором для специализированных приложений.

Исследование влияния характера данных на производительность показало, что кластеризованные ключи и данные с высокой коллизией могут приводить к деградации производительности в 2-3 раза по сравнению с оптимальными сценариями. Это подтверждает критическую важность выбора качественной хэш-функции и понимания природы обрабатываемых данных.

Сравнение с промышленной реализацией STL unordered_map выявило отставание кастомной реализации в 2-5 раз по различным операциям, что является ожидаемым результатом для учебного проекта. При этом кастомная реализация сохраняет предсказуемость времени отклика и демонстрирует достаточную для многих практических применений производительность.

Анализ "бутылочных горлышек" с использованием флеймграфов идентифицировал операцию рехэширования как основной источник накладных расходов, что соответствует теоретическим ожиданиям для структур данных с динамическим изменением размера.

В целом, реализация хэш-таблицы с открытой адресацией подтвердила свою эффективность для задач словаря, обеспечивая высокую производительность при условии правильного выбора метода пробирования в зависимости от характеристик входных данных.