Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования «Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики»

Факультет «Программной Инженерии и Компьютерных Технологий»

Отчет о лабораторной работе № 1

«Изучение производительности структуры»

по дисциплине

«Программирование на C++»

Выполнил:
Студент группы P4119
Суровикин В. Ю.
Проверил:
Доцент факультета ПИиКТ
Жданов А.Д.

Санкт-Петербург 2025

Задача

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.

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.

Вариант: 507549 mod 3 = 0 - Самобалансирующееся дерево поиска

AVL-дерево

Для реализации самобалансирующегося дерево была выбрана структура AVL-дерева. AVL-дерево представляет собой сбалансированное бинарное дерево поиска, где для каждого узла поддерживается строгий инвариант: разница высот левого и правого поддеревьев не превышает единицы. Эта самобалансирующаяся структура организована иерархически - каждый узел содержит ключ, значение, высоту поддерева и ссылки на

потомков. Ключевой механизм работы основан на системе поворотов, которые автоматически активируются при нарушении баланса после операций модификации. Благодаря поддержанию логарифмической высоты дерева, все основные операции гарантированно выполняются за O(log n) времени даже в наихудших сценариях, что обеспечивает предсказуемую производительность независимо от паттерна входных данных.

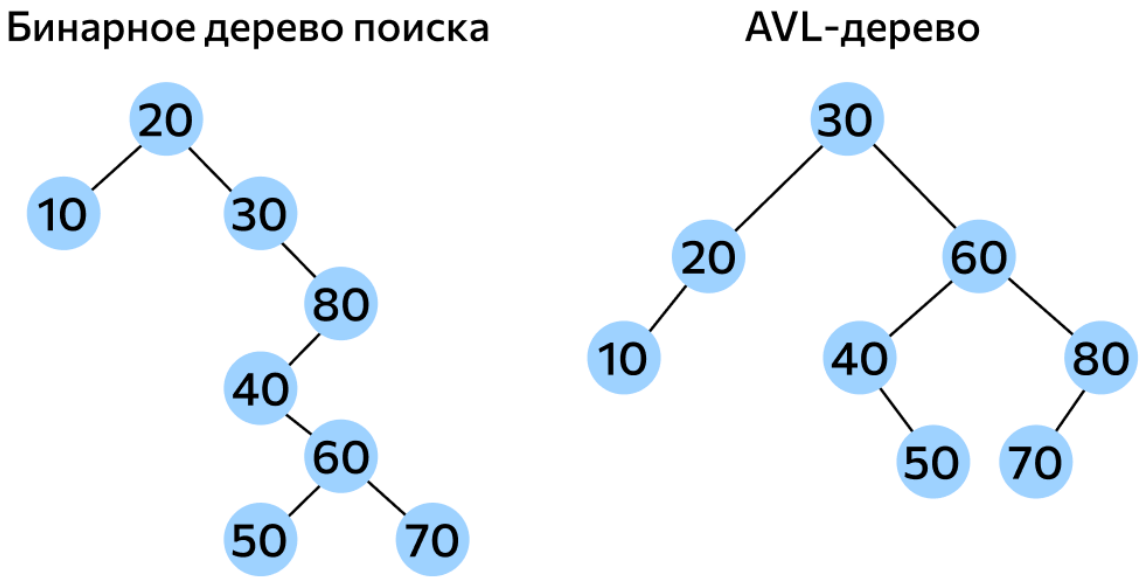

Рисунок 1 - Разница организаций обычного бинарного и AVL-деревьев

Реализация

Моя реализация AVL-дерева использует современные подходы C++ с применением умных указателей std::unique_ptr для автоматического управления памятью. Структура построена вокруг класса AVLNode, который инкапсулирует ключ, значение, высоту узла и владеющие указатели на потомков.

```cpp
class AVLNode {
 public:
  int key_;
  std::string value_;
  int height_;
  std::unique_ptr<AVLNode> left_;
  std::unique_ptr<AVLNode> right_;

  AVLNode(int k, const std::string &v)
      : key_(k), value_(v), height_(1), left_(nullptr), right_(nullptr) {}
};
```

Рисунок 2 - Узел AVL- дерева

Ключевые особенности реализации:

- **Безопасное управление памятью**: std::unique_ptr гарантирует автоматическое освобождение узлов, исключая утечки памяти
- **Рекурсивные алгоритмы**: Все операции (вставка, удаление, поиск) реализованы через рекурсивные функции, что естественно для древовидных структур
- **Двойная семантика модификации**: Поддержка как insert (только добавление), так и upsert (обновление существующих)
- **Комплексное удаление**: При удалении узла с двумя потомками используется стратегия замены на минимальный элемент правого поддерева
- **Сбалансированные повороты**: Реализованы все четыре типа поворотов (LL, RR, LR, RL) через комбинации rotateLeft и rotateRight

Архитектурные решения:

- Методы балансировки и обновления высот вынесены в отдельные функции
- Использование std::optional для безопасного возврата результатов поиска
- Рекурсивная перебалансировка поднимается от точки изменения к корню
- Вспомогательный метод popMin для извлечения минимального элемента при удалении

Бенчмарки

Для измерения времени задержек каждой операции использовался фреймворк Google Benchmark в паре с инструментами стандартной библиотеки chrono.

При тестировании операции вставки были выбраны три типа входных данных: последовательная данные, сбалансированные данные, случайные данные. Остальные типы операций проверялись на случайной выборке данных, так как входные данные уже никак не влияли на время задержки, дерево уже будет сбалансировано к моменту исполнения операций поиска и удаления.

Особое внимание уделялось минимизации влияния сторонних факторов на результаты измерений. Предварительный прогрев кэша выполнялся в отдельном бенчмарке, что обеспечивало стабильность последующих измерений. Перед запуском бенчмарков отключался TurboBoost, а также процессор переключался в режим performance для выставления частоты на максимальное и стабильное значение. Процессу задавался максимальный приоритет выполнения. Все временные замеры сохранялись в векторы и экспортировались в CSV-файлы для последующего статистического анализа и построения распределений вероятности, что позволило выявить не только средние значения, но и характеристики разброса, включая 99-й перцентиль задержек.

Графики распределения задержки

Анализ операции вставки в AVL-дерево выявил несущественные различия в производительности между тремя тестовыми сценариями, что напрямую связано с особенностями алгоритма балансировки.

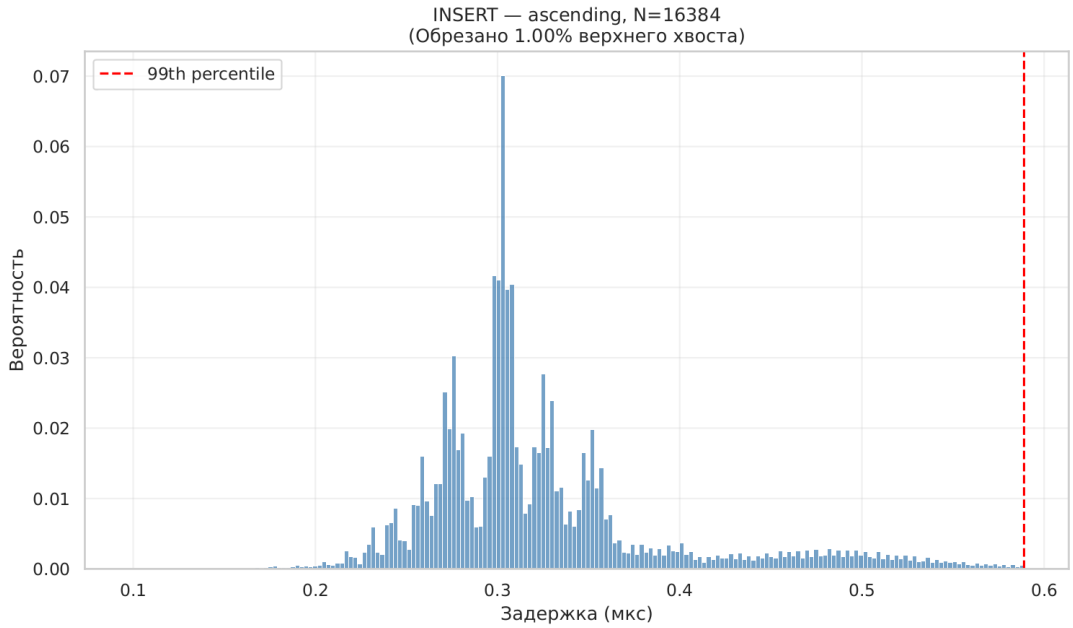

Рисунок 3 - График распределения задержек операции вставки для последовательных данных

Наилучшие показатели демонстрирует последовательный сценарий, где пик распределения задержек составляет около 0.3 микросекунд, а 99-й перцентиль не превышает 0.6 микросекунд. При последовательной вставке отсортированных данных дерево постоянно поддерживает баланс через однотипные правые повороты, которые выполняются достаточно эффективно. Однако периодически возникают более сложные случаи балансировки, что и приводит к удлинению правого хвоста распределения.

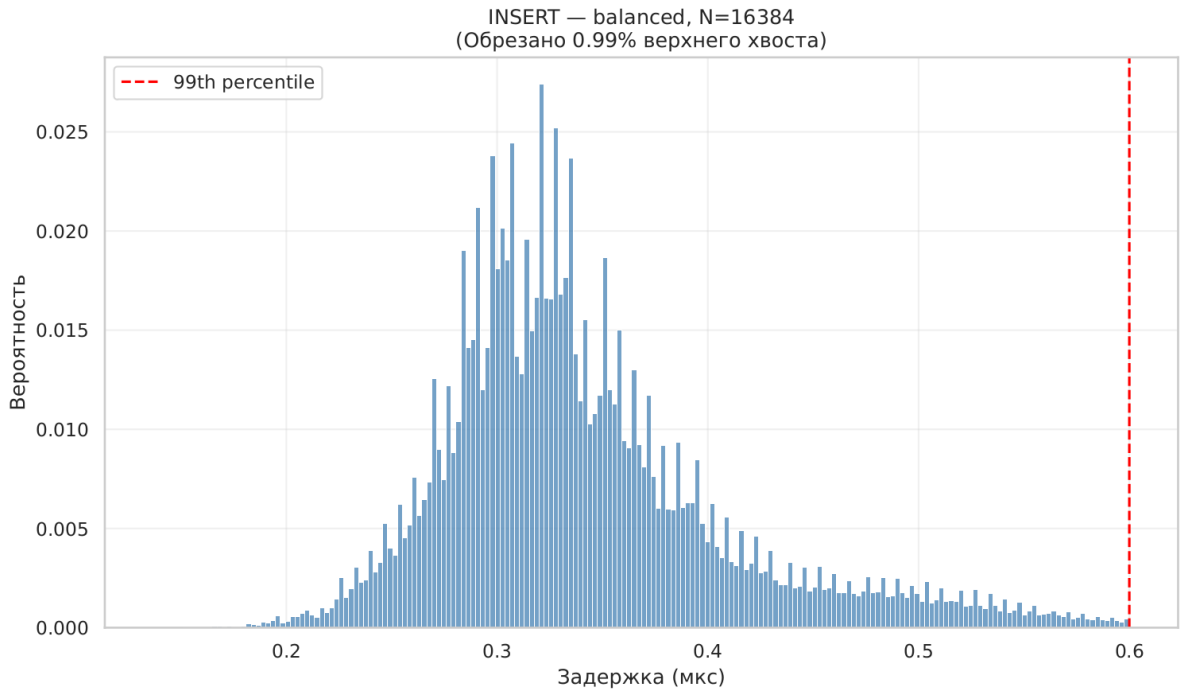

Рисунок 4 - График распределения задержек операции вставки для сбалансированных данных

Сбалансированный сценарий, показывает пик задержки на уровне 0.33 микросекунд, а 99-м перцентиль не превышает 0.6 микросекунд. Такой результат объясняется оптимальным порядком вставки элементов, при котором дерево изначально формируется с минимальной высотой и требует наименьшего количества поворотов для поддержания баланса. Рекурсивно сбалансированная последовательность вставки обеспечивает равномерное распределение узлов между левым и правым поддеревьями, минимизируя необходимость перестройки структуры.

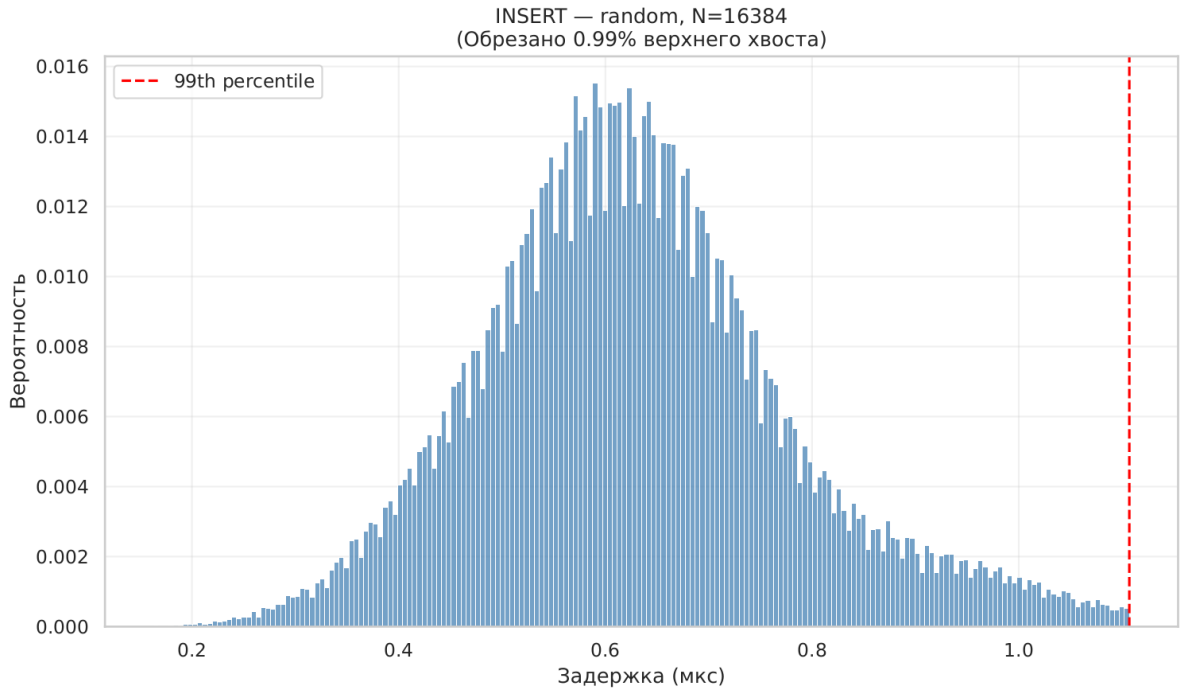

Рисунок 5 - График распределения задержек операции вставки для случайных данных

Наиболее интересные результаты демонстрирует сценарий random, где наблюдается самый широкий разброс задержек с пиком около 0.6 микросекунд и 99-м перцентилем до 1.5 микросекунд. Случайный порядок вставки приводит к непредсказуемой последовательности операций балансировки, требующих различных комбинаций поворотов - как простых, так и двойных. Это разнообразие в типах и сложности балансировочных операций объясняет значительный разброс в производительности. Кроме того, случайный доступ к памяти может вызывать промахи кэша, что дополнительно увеличивает задержки по сравнению с более предсказуемыми паттернами доступа в других сценариях.

Таким образом, разница в производительности между сценариями вставки наглядно демонстрирует ключевое преимущество AVL-дерева - способность эффективно противостоять деградации производительности в наихудших сценариях, сохраняя предсказуемое время выполнения операций даже при неблагоприятных паттернах данных.

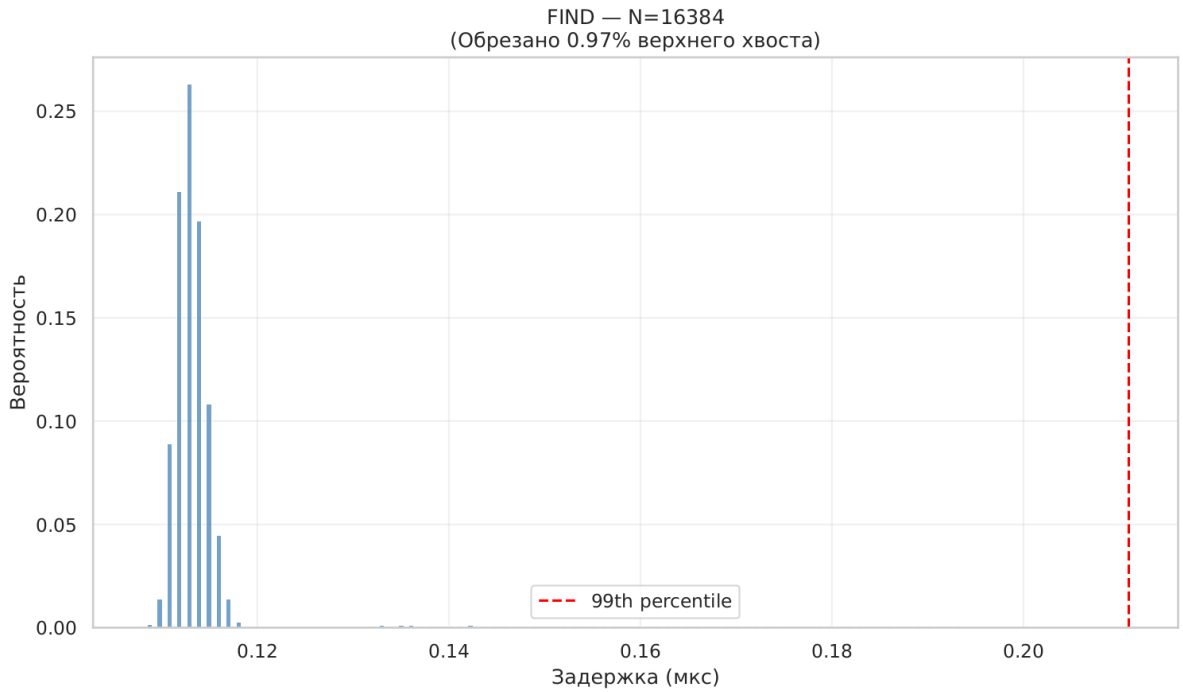

Рисунок 6 - График распределения задержек операции поиска

Операция поиска показывает наилучшие результаты с пиком распределения вокруг 0.11 микросекунд, что подтверждает логарифмическую сложность доступа даже для дерева из 16384 элементов.

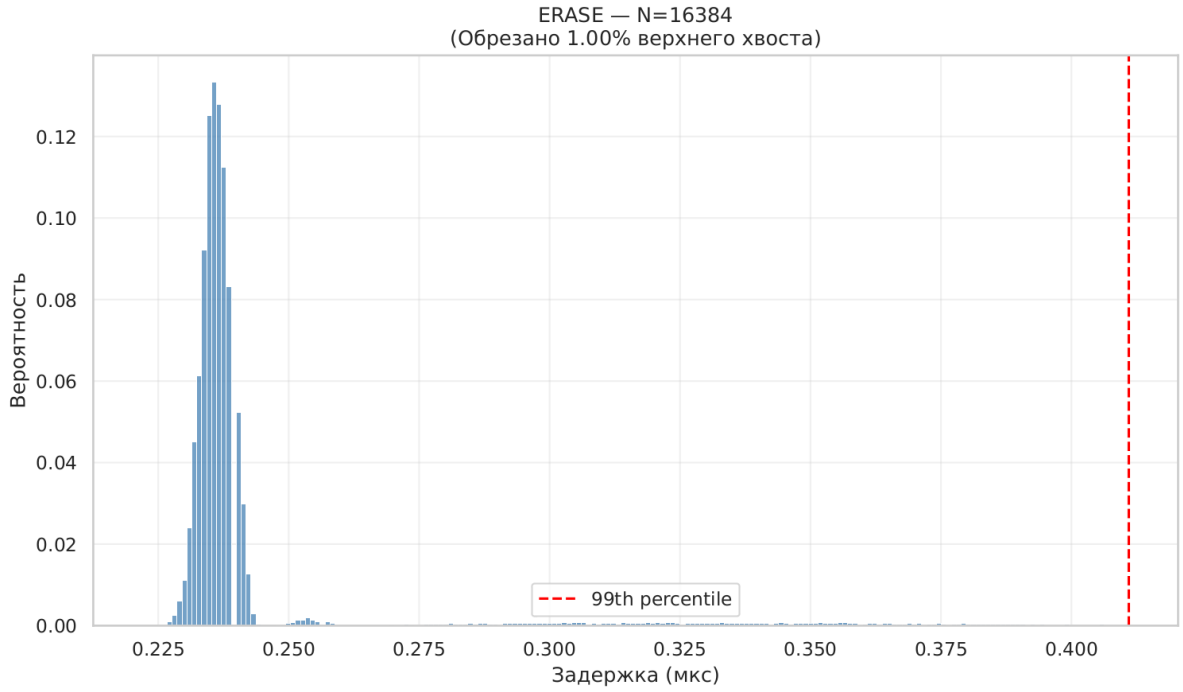

Рисунок 7 - График распределения задержек операции удаления

Удаление, являясь наиболее затратной операцией из-за необходимости перебалансировки, тем не менее укладывается в 0.23 микросекунды в пике, с 99-м перцентилем на уровне 0.42 микросекунд.

Распределения задержек всех операций характеризуются четко выраженными пиками и умеренными хвостами, что свидетельствует об отсутствии аномальных выбросов и стабильности работы структуры данных. Узкие пики для операций поиска и удаления подчеркивают предсказуемость их выполнения, в то время как более широкое распределение при вставке случайных данных отражает вариативность количества необходимых поворотов при балансировке.

Флеймграф

Для определения “узкого горлышка” программы, с помощью утилиты perf и метода сэмплирования построим флеймграф выполняемой программы.

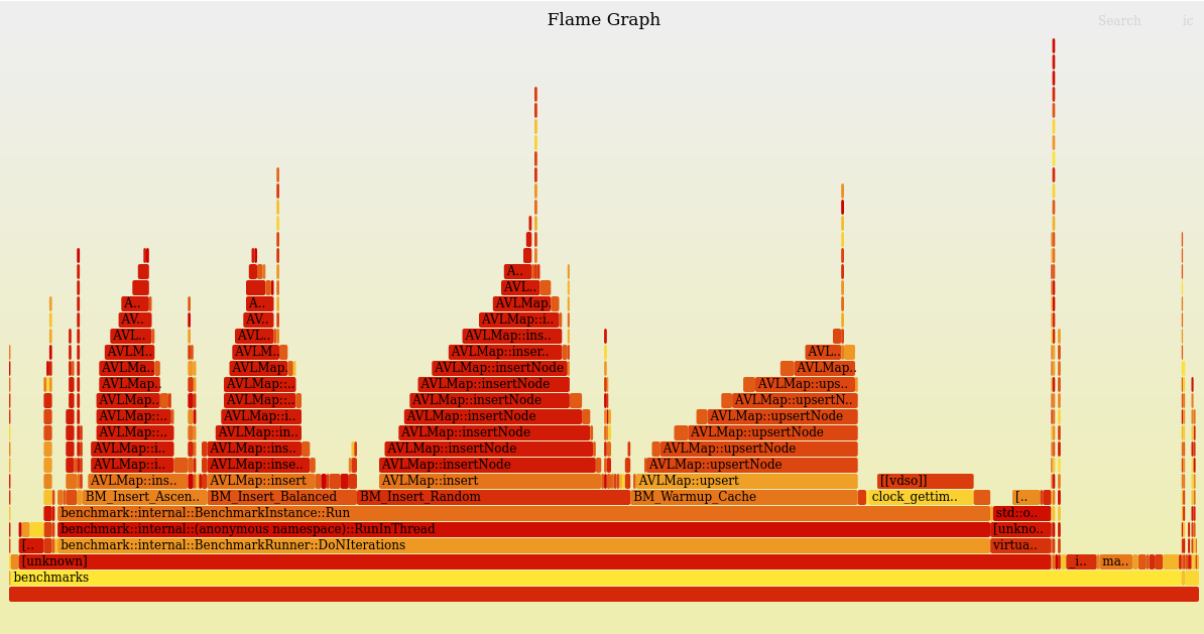

Рисунок 8 - Флеймграф бенчмарка

Данный флеймграф показывает “горячие места” всех проводимых бенчмарков.

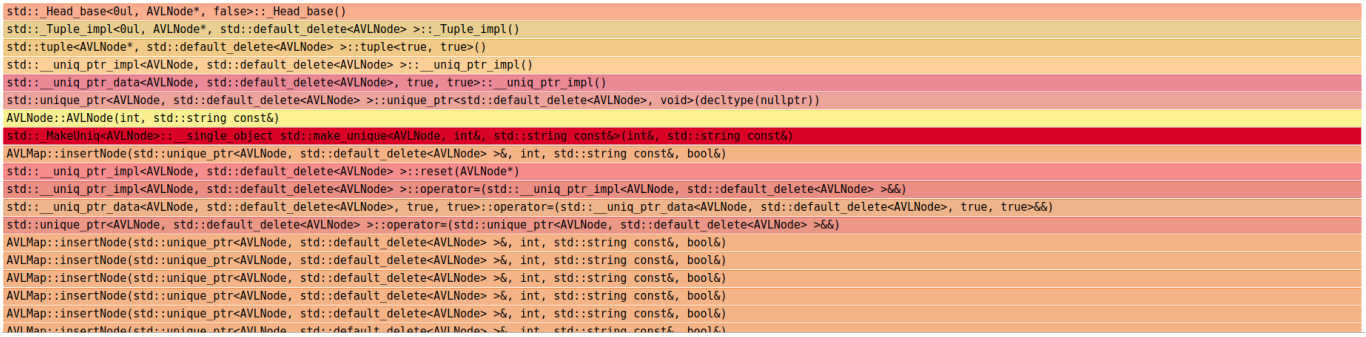

Рисунок 9 - “Узкое горлышко” программы

При детальном анализе флеймграфа я обнаружил, что самая “горячая” операция является операция создания умного указателя на узел. Это свидетельствует о том, что управление динамической памятью стало основным "узким горлышком" производительности. Обнаруженная проблема является классическим случаем, когда абстракции C++ стандартной библиотеки, хотя и обеспечивающие безопасность памяти, создают накладные расходы в performance-critical коде. Но данные накладные расходы не столь критичны, чтобы заменять безопасное использование памяти на использование сырых указателей. Дальнейшей оптимизации не требуется.

Вывод

Проведенное исследование позволило успешно реализовать и всесторонне проанализировать производительность словаря на основе AVL-дерева. Реализация демонстрирует стабильную логарифмическую сложность операций вставки, поиска и удаления, что подтверждается результатами бенчмарков для различных сценариев входных данных. Все операции показывают суб-микросекундные задержки даже для больших объемов данных (N=16384), что свидетельствует о высокой эффективности решения.