

Федеральное государственное автономное образовательное
учреждение высшего образования
Университет ИТМО

Факультет программной инженерии и компьютерной техники

**Отчёт по лабораторной работе №1
по дисциплине «Низкоуровневое
программирование»**

Вариант: Документное дерево

Выполнил:
Деев Роман Александрович

Группа: **P33102**

Преподаватели:
Кореньков Ю. Д.

Санкт-Петербург 2022 г.

Задание:

Описание заданий

Задание 1

Создать модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное обновление) информации общим объёмом от 10GB соответствующего варианту вида.

Порядок выполнения:

1. Спроектировать структуры данных для представления информации в оперативной памяти
 - a. Для порции данных, состоящий из элементов определённого рода (см форму данных), поддерживать тривиальные значения по меньшей мере следующих типов: четырёхбайтовые целые числа и числа с плавающей точкой, текстовые строки произвольной длины, булевские значения
 - b. Для информации о запросе
2. Спроектировать представление данных с учетом схемы для файла данных и реализовать базовые операции для работы с ним:
 - a. Операции над схемой данных (создание и удаление элементов схемы)
 - b. Базовые операции над элементами данных в соответствии с текущим состоянием схемы (над узлами или записями заданного вида)
 - i. Вставка элемента данных
 - ii. Перечисление элементов данных
 - iii. Обновление элемента данных
 - iv. Удаление элемента данных
3. Используя в сигнатурах только структуры данных из п.1, реализовать публичный интерфейс со следующими операциями над файлом данных:
 - a. Добавление, удаление и получение информации о элементах схемы данных, размещаемых в файле данных, на уровне, соответствующем виду узлов или записей
 - b. Добавление нового элемента данных определённого вида
 - c. Выборка набора элементов данных с учётом заданных условий и отношений со смежными элементами данных (по свойствам/полям/атрибутам и логическим связям соответственно)
 - d. Обновление элементов данных, соответствующих заданным условиям
 - e. Удаление элементов данных, соответствующих заданным условиям
4. Реализовать тестовую программу для демонстрации работоспособности решения
 - a. Параметры для всех операций задаются посредством формирования соответствующих структур данных
 - b. Показать, что при выполнении операций, результат выполнения которых не отражает отношения между элементами данных, потребление оперативной памяти стремится к $O(1)$ независимо от общего объёма фактического затрагиваемых данных
 - c. Показать, что операция вставки выполняется за $O(1)$ независимо от размера данных, представленных в файле
 - d. Показать, что операция выборки без учёта отношений (но с опциональными условиями) выполняется за $O(n)$, где n – количество представленных элементов данных выбираемого вида
 - e. Показать, что операции обновления и удаления элемента данных выполняются не более чем за $O(n \cdot m) > t \rightarrow O(n+m)$, где n – количество представленных элементов данных обрабатываемого вида, m – количество фактически затронутых элементов данных
 - f. Показать, что размер файла данных всегда пропорционален количеству фактически размещённых элементов данных
 - g. Показать работоспособность решения под управлением ОС семейств Windows и *NIX
5. Результаты тестирования по п.4 представить в составе отчёта, при этом:
 - a. В части 3 привести описание структур данных, разработанных в соответствии с п.1
 - b. В части 4 описать решение, реализованное в соответствии с пп.2-3
 - c. В часть 5 включить графики на основе тестов, демонстрирующие амортизированные показатели ресурсоёмкости по п. 4

Цель

Реализовать базу данных хранящую своё состояние в файле на жёстком диске. Элементы в базе данных должны быть представлены в виде документного дерева. Помимо этого, операции с элементами данных должны удовлетворять ограничениям по времени и памяти.

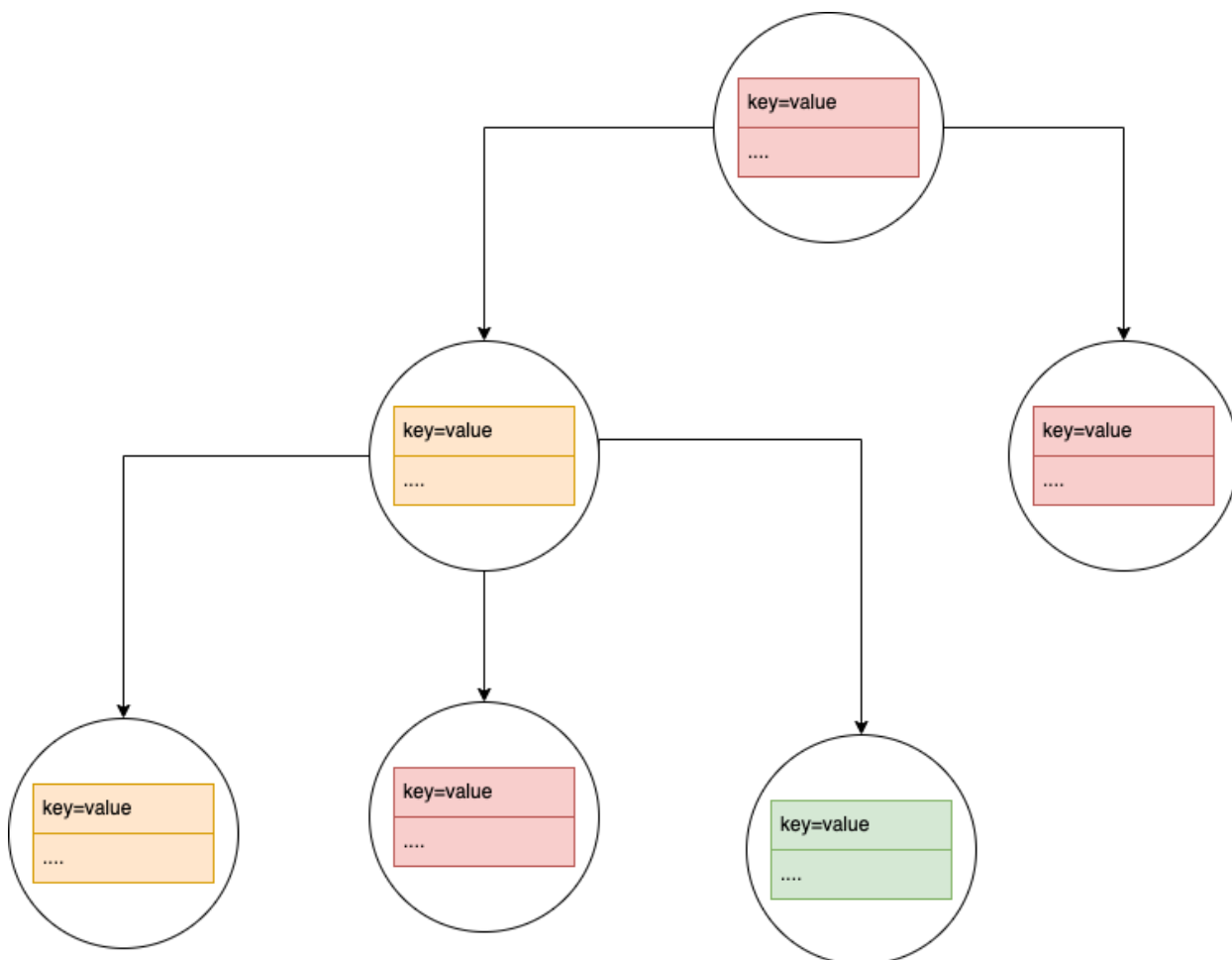
Идея решения

Основная проблема, с которой придётся бороться — переиспользование высвободившегося места от удалённых элементов, которые могут быть различной длины.

Для борьбы с этим сделаем любые данные, гранулярными: будем записывать их кусками одинаковой длины (далее — чанки), но в каждом из которых будет ссылка на следующий элемент. Получаем, что мы можем разбить весь файл на чанки. Незанятые чанки объединим в связанный список. При записи порции данных возьмём несколько чанков из списка свободных, если список пуст — расширим файл для выделения пачки чанков (страницы). Высвобождаемые чанки возвращаем в список свободных.

Мы получили интерфейс для записи порции данных (кучки байт) в файл, теперь поверх выстроим логику поддержания документного дерева.

Визуализация структуры дерева. Цветами обозначены схема, к которой относится узел



Выделим кучки байт, которые мы ходим записывать:

1. Схемы
2. Узлы дерева
3. Строки

Строки

Любые строки храним в виде указателя (смещения в файле) на первый байт.

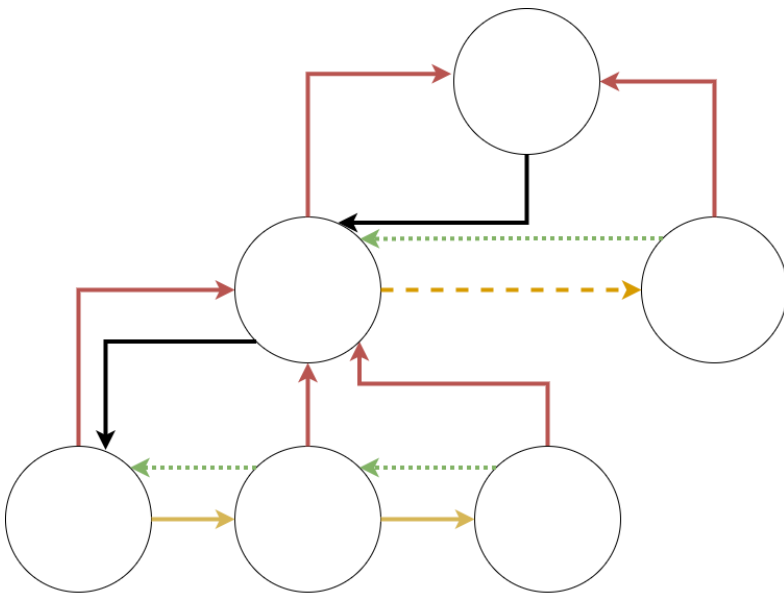
Схемы

Храним в виде связанного списка. Т.е. помимо информации о допустимых ключах и значениях, храним место в файле со следующей схемой.

Узлы дерева

Каждому узлу в дереве при добавлении назначаем id, который под капотом будет указателем на первый чанк узла. Приходящие id от пользователя придётся проверять на валидность.

Введение id даёт нам вставку за $O(1)$ от количества элементов: на вход достаточно принять id предка в дереве. Так как id это указатель место элемента, то найти его для записи нового сына сможем за $O(1)$



Помимо ссылки на предка (**красная**) в объекте узла будем хранить ссылку первого сына(**чёрная**), брата (**жёлтая**), предыдущего брата (**зелёная**) и массив ключей/значений. Ключ — ссылка на строку. Значение — union от int32, double, bool, и ссылки на строку. Таким образом обновление значений элемента никак не повлияет на расположение узла в памяти, так как размер элемента не изменяется при таком подходе.

Поиск элементов по условию — обход дерева $O(n)$

Для удаления по условию за $O(n)$ придётся реализовать обратный обход (LRN), потому что возможна ситуация, что узел невозможно удалить, пока не удалены его потомки. Удаление конкретного элемента занимает $O(1)$ так как известна его позиция в файле и все ссылающиеся на него элементы.

Аспекты реализации

Главный класс — Database.

Через его публичные методы происходит выполнение требуемых операций с деревом. Принимаемые аргументы и возвращаемые значение описаны в Query.h.

Для чтения записи произвольные байтов Database есть объект FileInterface с методами Write/Read

Методы содержащие в названии BytesBatch служат для записи/чтения/удаления чанков объектов.

Остальные методы -- вспомогательные для публичных методов и итераторов по схемам и по дереву.

Далее можно выделить три типа классов:

1. Классы объектов базы отдаваемые пользователю: Element, Schema, ...
2. Упакованные классы служащие которые будут записываться BytesBatch-методами: raw_*, ...
3. Промежуточные между 1 и 2: ElementBox, SchemaBox. С ними взаимодействуют вспомогательные методы

Результаты

Исходный код: <https://github.com/deevroman/low-level-programming-labs>

Код реализованного модуля лежит в папке llp.

Модуль представляет собой stake INTERFACE, пример использования можно найти в папке examples.

Больше примеров работы с публичными методами можно найти в папке llp/tests. В файле bench.cpp код бенчмарков.

Сборка и тестирование поддержаны для Ubuntu (g++, clang), macOS (clang), Windows (msbuild, mingw32)

Для сборки у себя:

```
git clone https://github.com/deevroman/low-level-programming-labs.git && cd low-level-programming-labs
# сборка модуля
cmake -B llp/build -DCMAKE_BUILD_TYPE=Release llp
cmake --build llp/build --config Release
# сборка примера
cmake -B examples/build -DCMAKE_BUILD_TYPE=Release examples
cmake --build examples/build --config Release
# запуск
./examples/build/bin/*/*
```

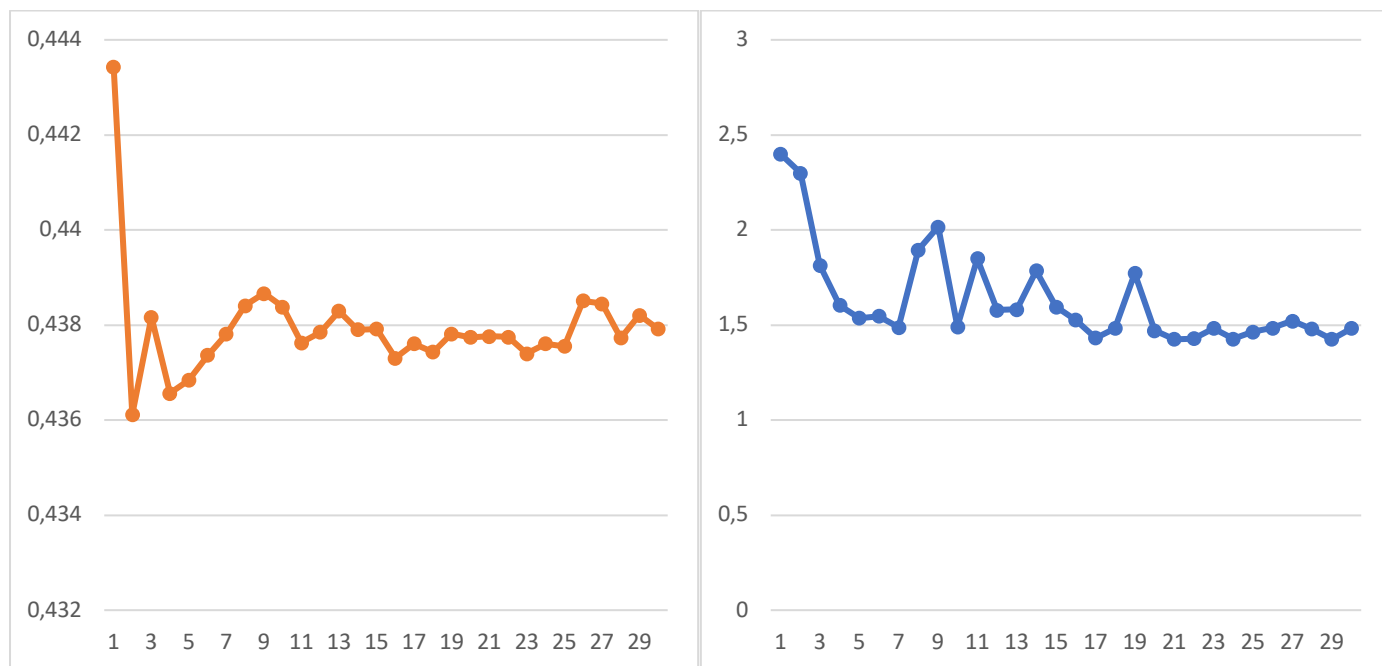
Бинарник после запуска должен вывести: Usage: ./example path/to/file↵

При проблемах обратитесь к файлу `.github/workflows/сmake.yml` и к логам GitHub Action, в котором выполняются шаги описанные в файле.

Ресурсоёмкость

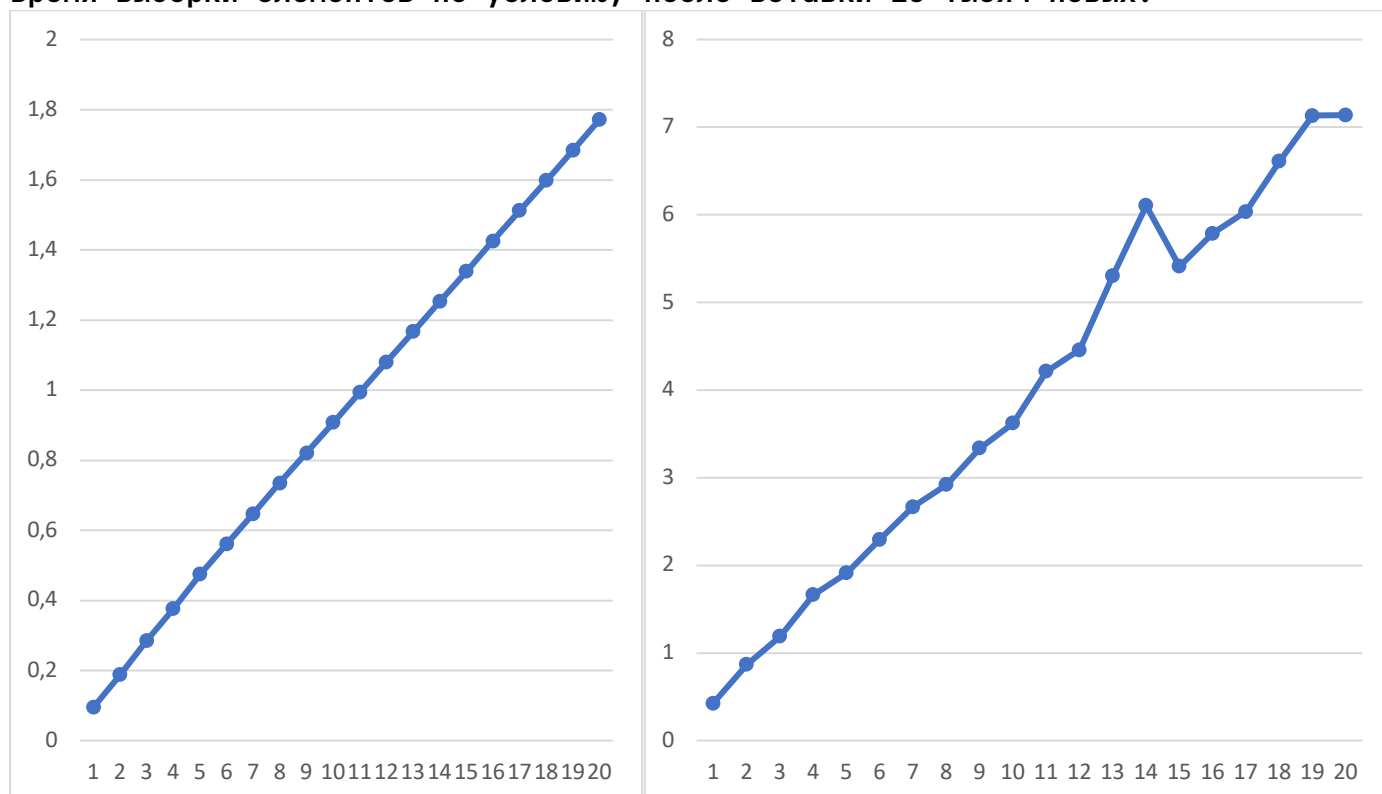
Слева результаты в GitHub Action, справа — на своём компьютере.

Время последовательной вставки 10 тысяч элементов:



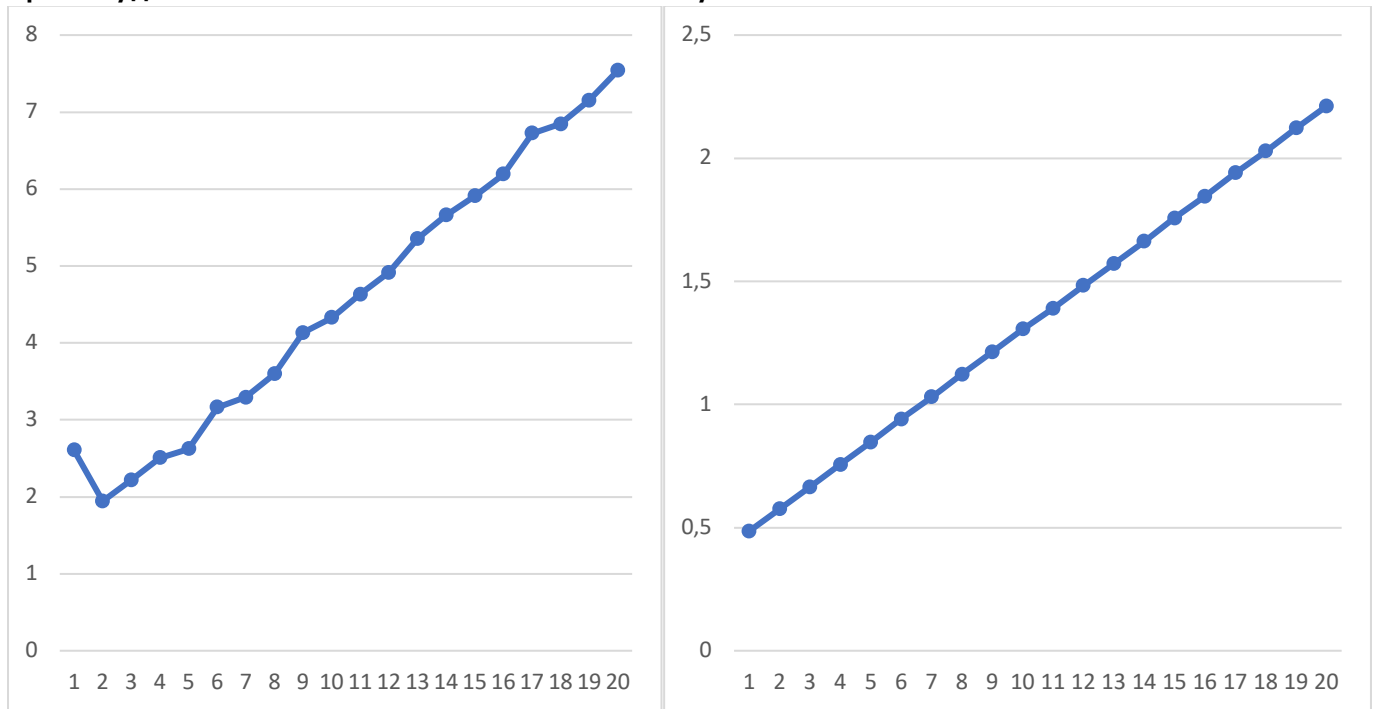
Вставка $O(1)$

Время выборки элементов по условию, после вставки 10 тысяч новых:



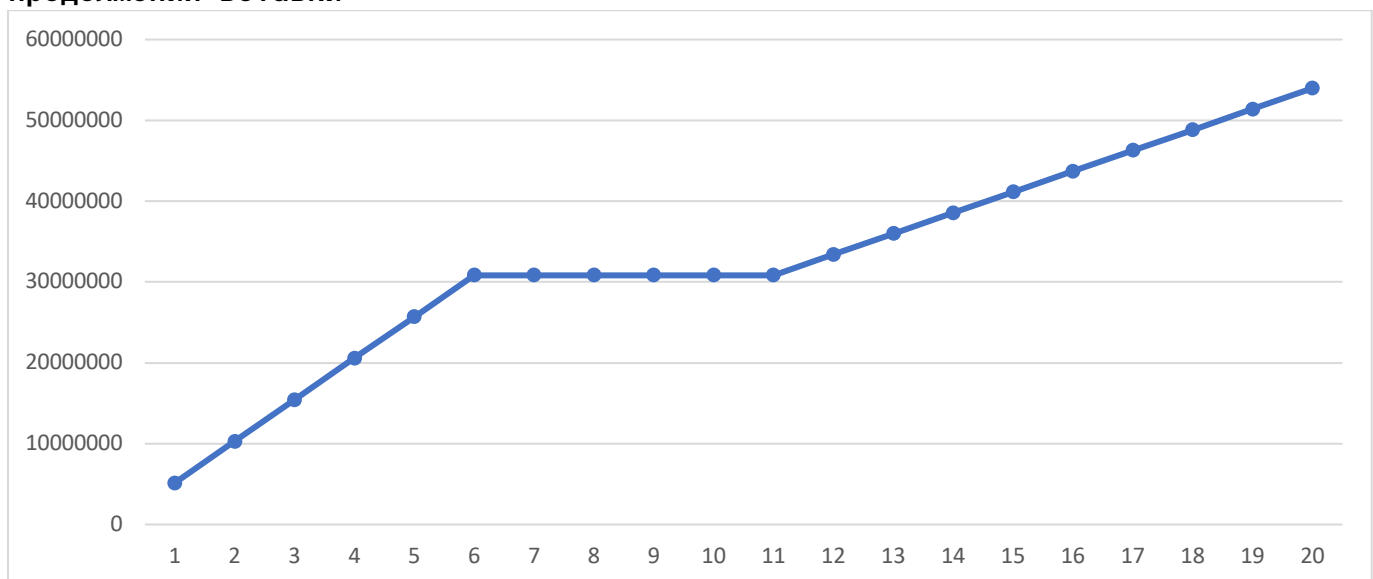
Поиск $O(n)$

Время удаления 5 тысяч элементов после увеличения элементов на 10 тысяч



Удаление $O(n)$

Размер файла в байтах после вставки 100 тысяч элементов, удаления половины, и продолжения вставки



Освобождаемое пространство переиспользуется.

Выводы:

- Реализовал хранение документного дерева в файле
- Провёл замеры времени работы
- Изучил фреймворк GoogleTest
- Попрактиковался в сборке на под разные платформы с помощью CMake и запуске тестов на разных платформах в GitHub Actions
- Изучил средства просмотра бинарных файлов и кастомные визуализаторы в отладчиках
- Возненавидел Windows и g++