САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ ИТМО

Дисциплина: Низкоуровневое программирование

Отчет

к практическому заданию №1

**«Модуль хранения данных»**

Выполнила:

Чернова Елизавета Александровна

Группа P33302

Санкт-Петербург

2023

**Цель работы:** создать модуль, реализующий хранение в одном файле данных (выборку, размещение и гранулярное обновление) информации общим объёмом от 10GB, представленной в виде документного древа.

**Инструментарий и требования к работе:** язык Cи.

Документное дерево — способ хранения данных, при котором данные хранятся иерархически, при этом внутренние вершины дерева выступают в роли директорий, а листья дерева несут какие-то данные.

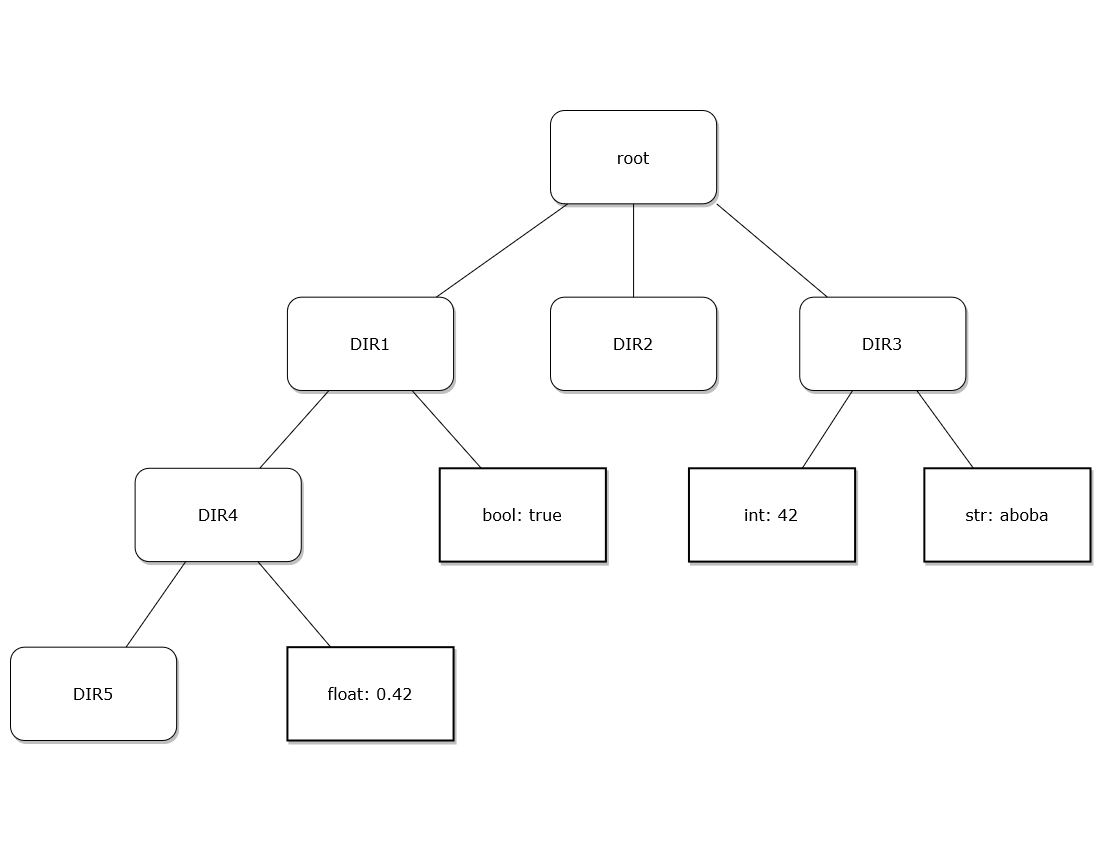


Рисунок 1 — форма хранения данных документное дерево.

**Задачи и план выполнения**

1. Спроектировать структуры данных для представления в оперативной памяти элементов (вершин) документного древа, при этом поддерживать возможность сохранять в листах дерева данные следующих типов: четырёхбайтовые целые числа и числа с плавающей точкой, текстовые строки произвольной длины, булевские значения.
2. Продумать способ хранения данных и директорий (внутренних вершин дерева) на диске.
3. Написать интерфейс взаимодействия с базой данных, в котором будут поддерживаться по меньшей мере следующие операции:
   1. создание и удаление директорий (папок/внутренних вершин дерева);
   2. создание и удаление листьев, несущих данные;
   3. обновление данных в существующем листе дерева;
   4. получение значений из листьев дерева;
   5. перечисление содержимого директории (всех детей вершины).
4. Написать тесты, использующие публичный интерфейс БД, и проверяющие корректность её работы.
5. Реализовать интерфейс из п.3.
6. Написать бенчмарки (тесты, измеряющие производительность решения).
7. Оформить отчёт, к которому приложить результаты бенчмарков.

**Интерфейс решения**

Решение состоит из заголовочных файлов, хранящихся в директории include/, файлов с реализацией (src/), тестов и бенчмарков (tests/). Проект использует **C17** и собирается с помощью **CMake 3.22**.

Все операции по созданию, наполнению, использованию и очистке файла с данными выполняются путём вызова соответствующих функций на языке Си. Интерфейс для работы с БД:

Database\* database\_create\_database(char const\* filename);

void database\_shutdown\_database(Database\* ptr); // does not erase any data

void database\_destroy\_database(Database\* ptr); // clears all data before shutting down

Directory\* database\_create\_directory(Database\* db, Directory\* parent, char const\* name);

bool database\_delete\_directory(Database\* db, Directory\* ptr);

void database\_clear\_directory(Database\* db, Directory\* dir);

Leaf\* database\_create\_leaf(Database\* db, Directory\* parent, char const\* name,

Types type, Value value);

bool database\_update\_leaf(Database\* db, Leaf\* leaf, Value new\_value);

Value const\* database\_get\_leaf\_value(Database const\* db, Leaf const\* leaf);

bool database\_delete\_leaf(Database\* db, Leaf\* ptr);

Iterator database\_get\_directory\_content\_iterator(Database const\* db, Directory const\* dir);

Пример использования:

Database\* db = database\_create\_database("example");

Value v;

v.int\_value = 42;

Leaf\* l1i = database\_create\_leaf(db, NULL, "L1i", INT, v);

fprintf(stderr, "Created L1i. null: %d, value: %d\n", l1i == NULL,

(l1i == NULL ? -1 : database\_get\_leaf\_value(db, l1i)->int\_value));

char\* s = "abc";

String ss;

ss.size = 3;

ss.data = s;

v.str\_value = ss;

Leaf\* l1s = database\_create\_leaf(db, NULL, "L1s", STR, v);

fprintf(stderr, "Created L1s. null: %d, value: %s\n", l1s == NULL,

(l1s == NULL ? "null" : database\_get\_leaf\_value(db, l1s)->str\_value.data));

Directory\* l1d = database\_create\_directory(db, NULL, "L1d");

fprintf(stderr, "Created L1d. null: %d\n", l1d == NULL);

Directory\* l2d1 = database\_create\_directory(db, l1d, "L2d1");

fprintf(stderr, "Created L2d1. null: %d\n", l2d1 == NULL);

Directory\* l2d2 = database\_create\_directory(db, l1d, "L2d2");

fprintf(stderr, "Created L2d2. null: %d\n", l2d2 == NULL);

v.float\_value = 0.42f;

Leaf\* l2f = database\_create\_leaf(db, l1d, "L2f", FLOAT, v);

fprintf(stderr, "Created L2f. null: %d, value: %f\n", l2f == NULL,

(l2f == NULL ? -1.0f : database\_get\_leaf\_value(db, l2f)->float\_value));

Directory\* l3d = database\_create\_directory(db, l2d1, "L3d");

fprintf(stderr, "Created L3d. null: %d\n", l3d == NULL);

fprintf(stderr, "Finished setup.\n");

database\_traverse\_and\_print\_database(db);

bool res = database\_delete\_directory(db, l3d);

fprintf(stderr, "Deleted L3d: %d\n", res);

database\_traverse\_and\_print\_database(db);

res = database\_delete\_leaf(db, l2f);

fprintf(stderr, "Deleted L2f: %d\n", res);

database\_traverse\_and\_print\_database(db);

database\_destroy\_database(db);

Результат выполнения:

Created L1i. null: 0, value: 42

Created L1s. null: 0, value: abc

Created L1d. null: 0

Created L2d1. null: 0

Created L2d2. null: 0

Created L2f. null: 0, value: 0.420000

Created L3d. null: 0

Finished setup.

Traversing database...

|-- L1d:

|-- L2f=0.420000

|-- L2d2:

empty dir

|-- L2d1:

|-- L3d:

empty dir

|-- L1s=abc (len=3)

|-- L1i=42

Deleted L3d: 1

Traversing database...

|-- L1d:

|-- L2f=0.420000

|-- L2d2:

empty dir

|-- L2d1:

empty dir

|-- L1s=abc (len=3)

|-- L1i=42

Deleted L2f: 1

Traversing database...

|-- L1d:

|-- L2d2:

empty dir

|-- L2d1:

empty dir

|-- L1s=abc (len=3)

|-- L1i=42

**Архитектура решения**

Чтобы достичь требуемых асимптотик на вставку, удаление и обновление элементов (O(1)), вершины дерева будут устроены следующим образом: в каждой будет храниться указатель на следующего и предыдущего соседей на текущем уровне, а также если текущая вершина — директория, будем хранить указатель на ребёнка. Если же текущая вершина — лист, то вместо указателя на ребёнка будем хранить данные.

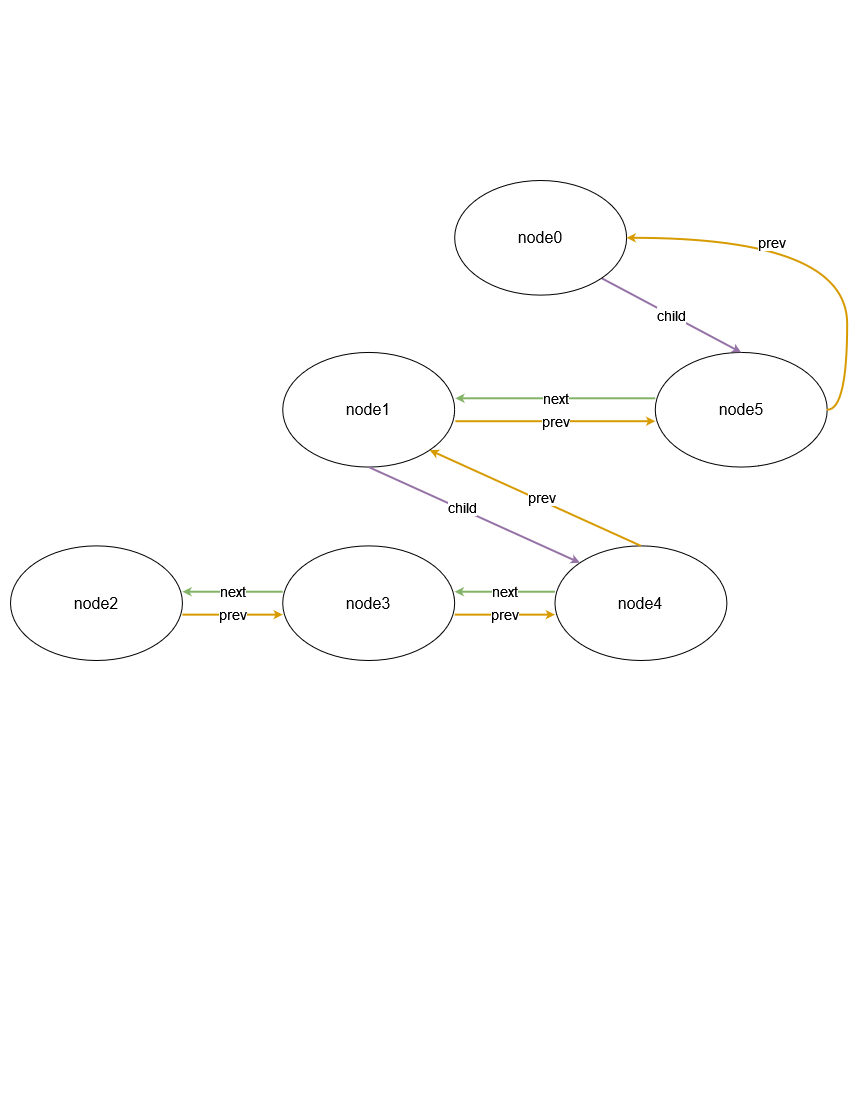


Рисунок 2 — связи между вершинами в дереве.

Таким образом, для вставки вершины достаточно будет лишь установить указатель родителя на новую вершину, а в новой вершине присвоить указатель next на текущего ребёнка. Что и гарантирует выполнение операций вставки за О(1).

Возникает вопрос: зачем дети внутри каждой директории состоят в двусвязном списке? Ответ: для быстрого удаления. Так как при удалении вершины нужно скорректировать указатель next в предыдущем соседе, то придётся хранить указатель prev.

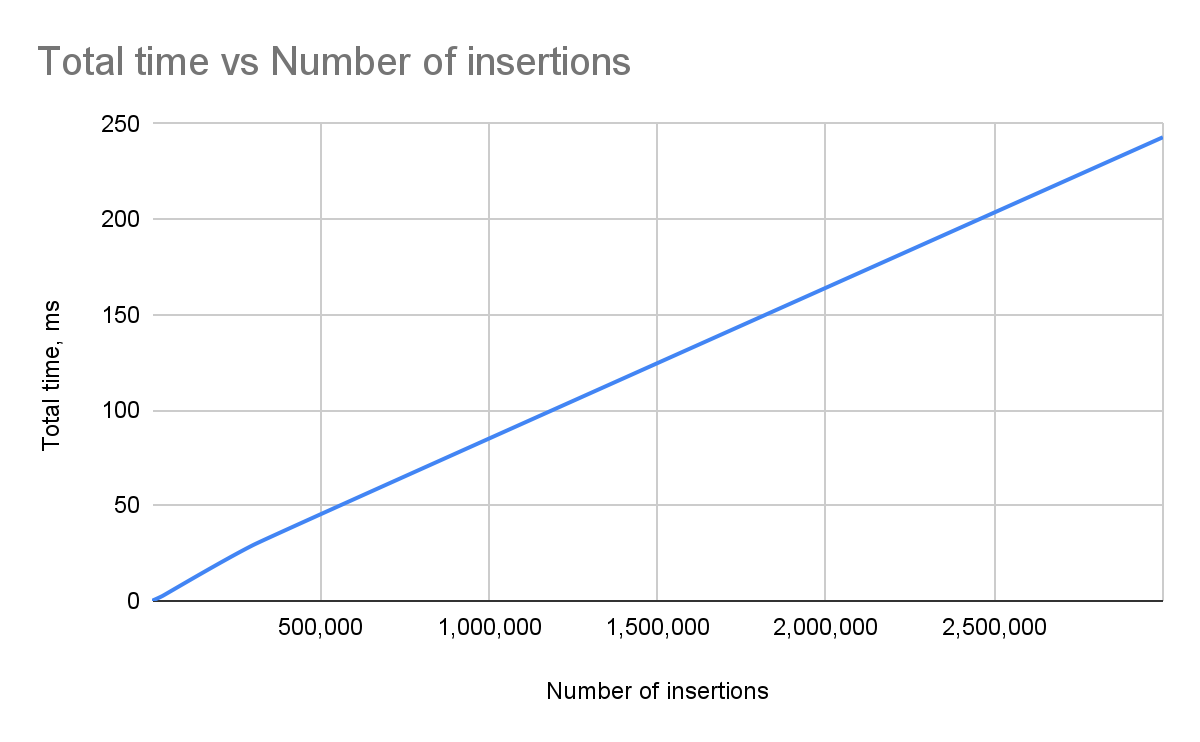
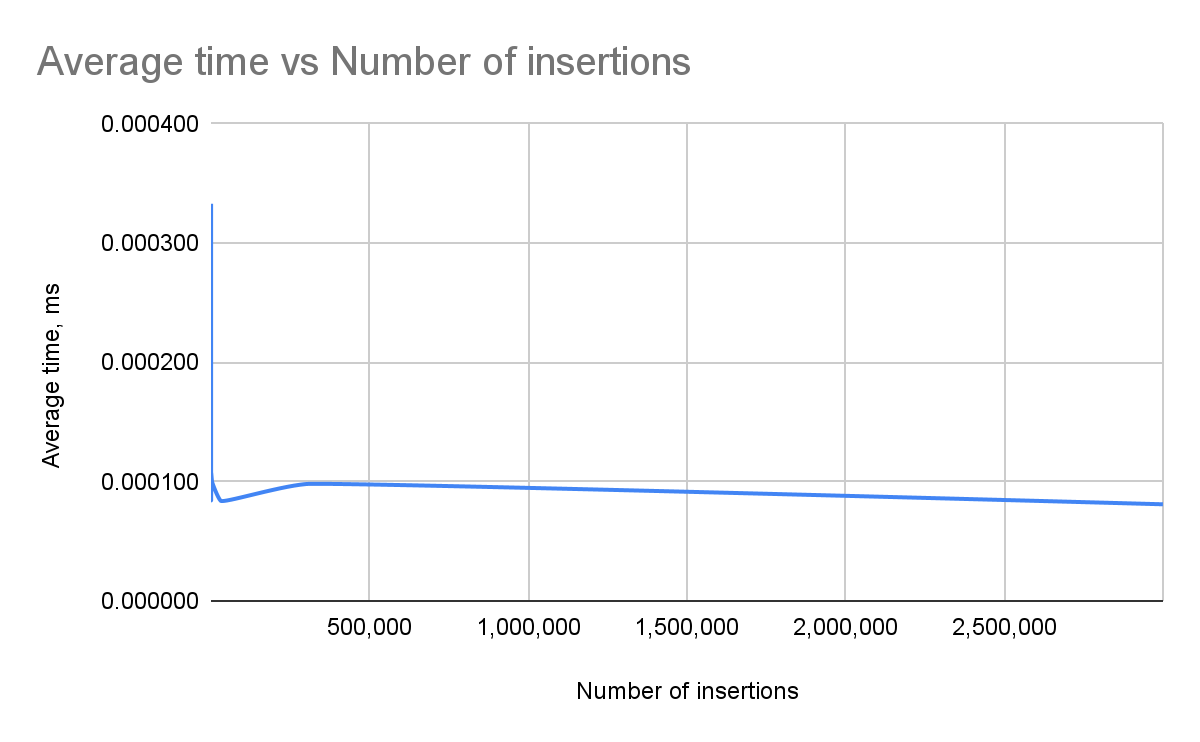
Заметим, что для оптимизации затрачиваемой памяти в вершине не хранится указатель на родителя. Родитель нужен в одном случае — когда мы удаляем ребёнка, который был добавлен последним. В такой ситуации будем в неиспользуемый указатель prev последнего ребёнка записывать адрес родителя. Таким образом экономится целых 8 байтов на вершину (на современных системах).

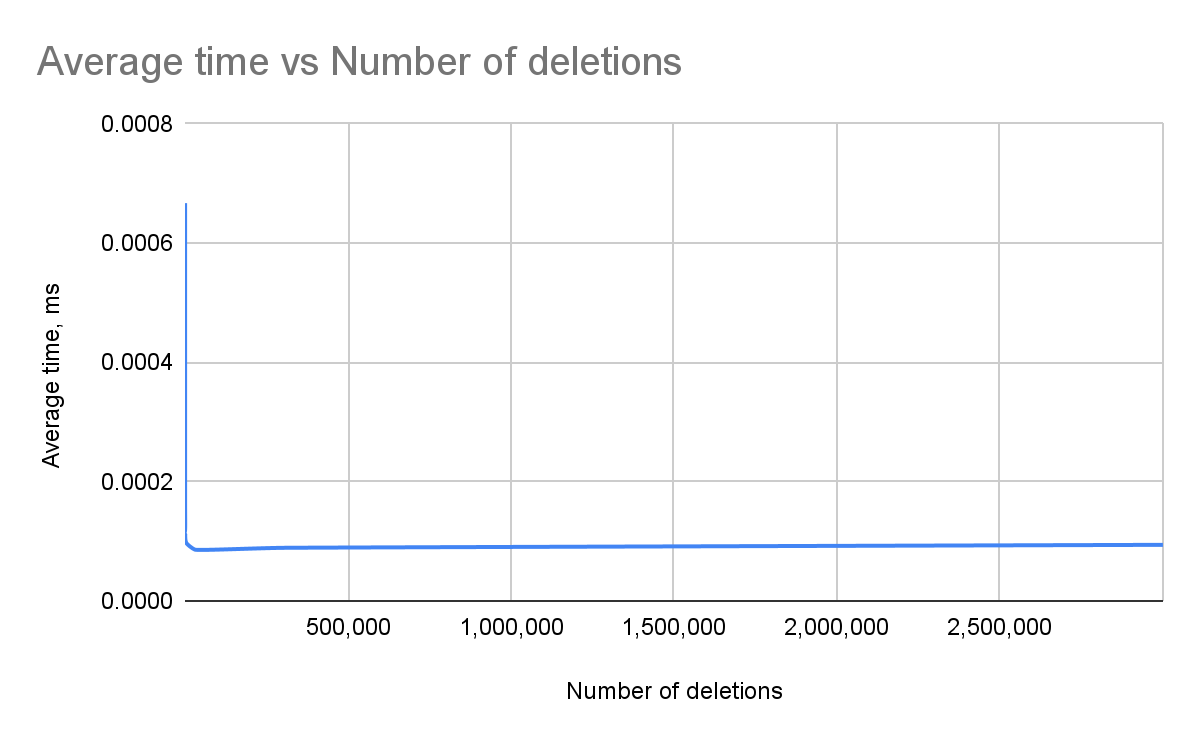
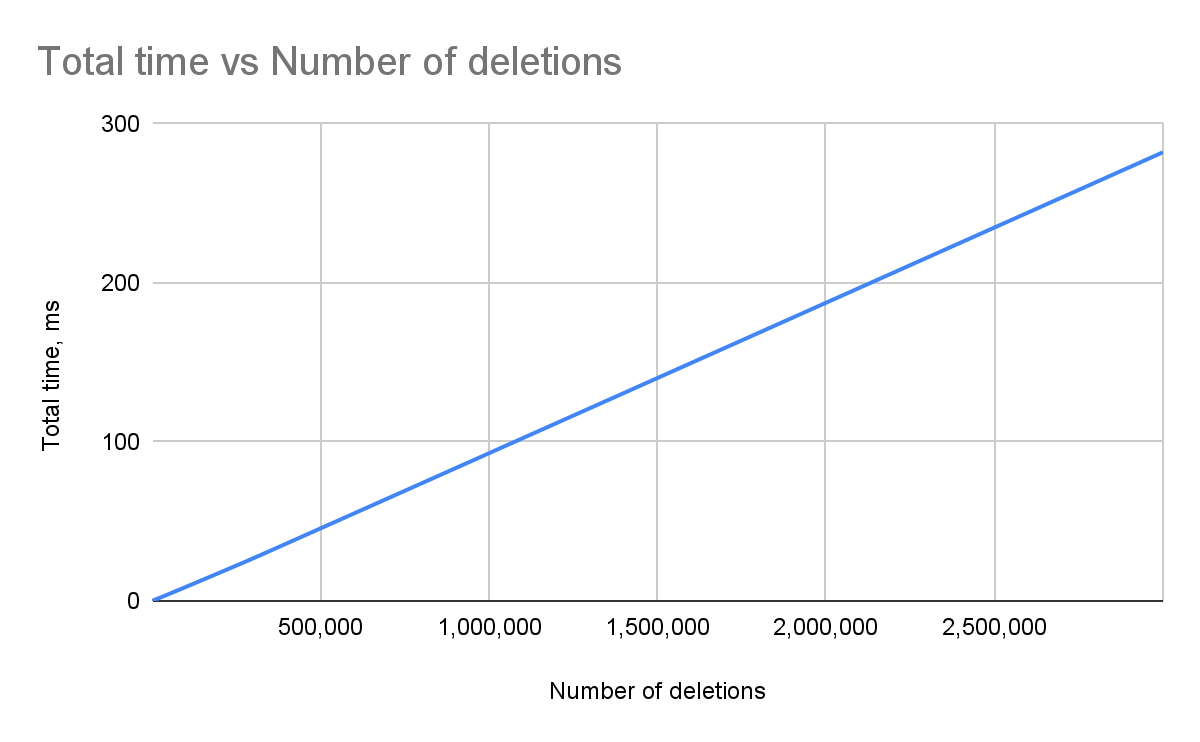
К слову о памяти. Изначально при проектировании интерфейса БД возник вопрос: реализовывать БД со схемой данных или без неё. Было решено предоставить пользователю бо́льшую гибкость и не требовать схемы данных. В таком случае становятся невозможны некоторые оптимизации по хранению данных (например, в каждой вершине приходится хранить указатели на следующую и предыдущую, а ещё не получится хранить имена вершин отдельно от данных (что можно было бы использовать, например, для быстрого выполнения агрегационных запросов к данным)). Однако пользователь не ограничен жёсткими рамками вида “в этой директории могут храниться только элементы типа X”, и может более свободно выстраивать удобную ему иерархическую структуру. Также из преимуществ отсутствия строгой схемы данных можно выделить относительную лёгкость в модифицировании текущего решения, если к нему захочется добавить какие-то оптимизации — поддержка индексов/кэшей, или хранение хэш-таблиц для более быстрого поиска вершин по пути.

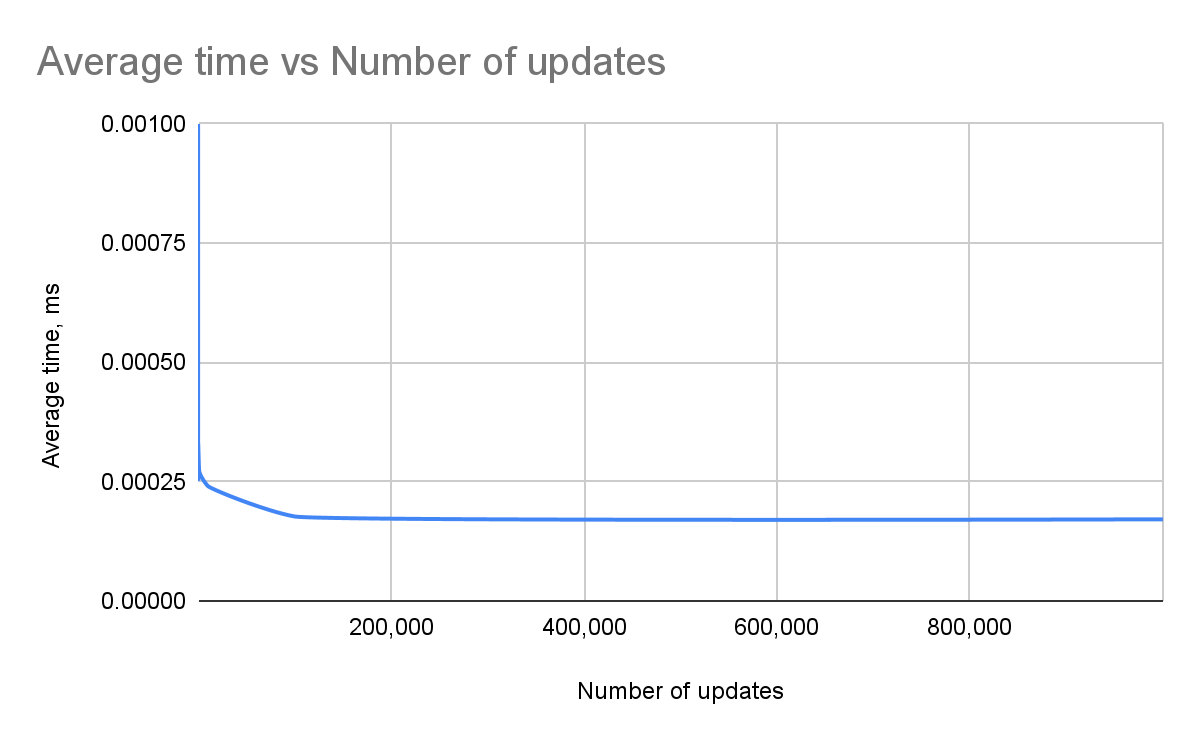
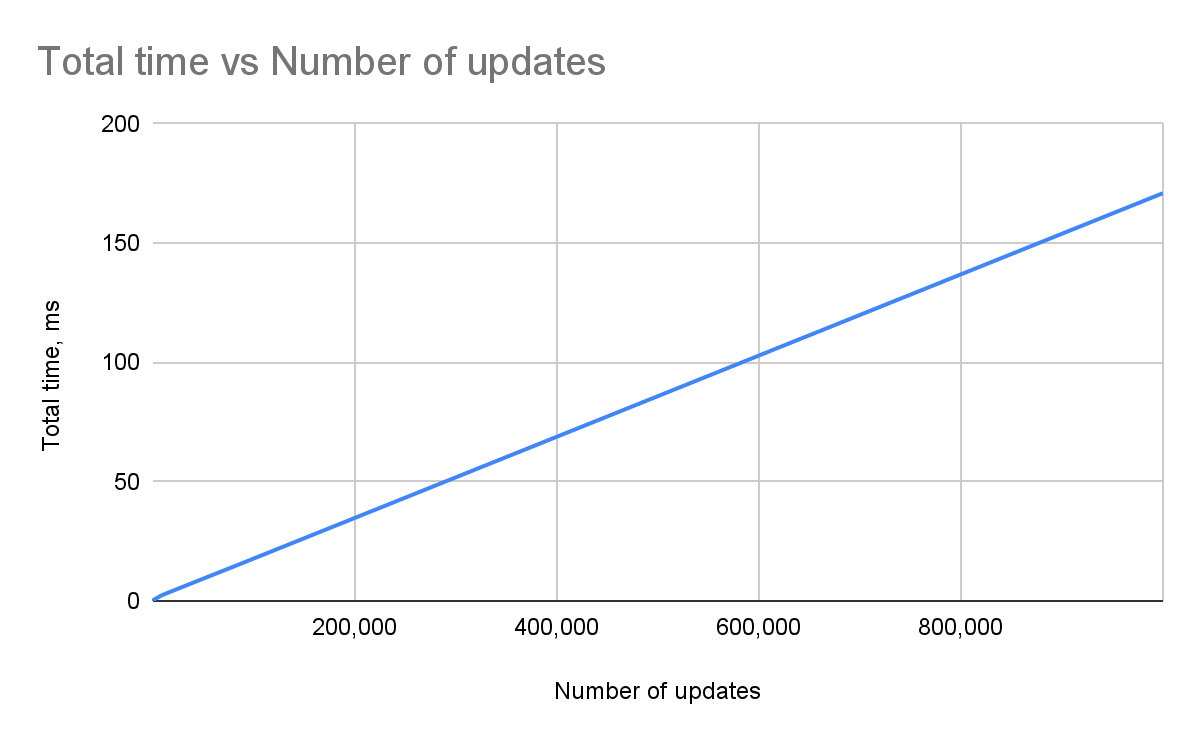
Для хранения данных, директорий и вспомогательной информации на диске (сами структуры Node тоже необходимо хранить на диске, т.к. их может быть настолько много, что они не поместятся в оперативную память) будем использовать свой аллокатор, который будет управлять свободным местом в файле (и, при необходимости, увеличивать файл в размере). Свой аллокатор позволит переиспользовать место, освободившееся после удаления каких-то вершин (и, таким образом, поддерживать инвариант линейной зависимости потребляемого места на диске от количества элементов в документном дереве). В качестве аллокатора будем использовать [buddy allocator](https://en.wikipedia.org/wiki/Buddy_memory_allocation) — простой в реализации, при этом относительно эффективный. Работать аллокатор будет через прослойку в виде memory mapping’а, которую требовалось использовать в моём варианте. Также memory mapping позволит работать с обычными адресами, а не с offset’ами в файле на диске, что упрощает реализацию аллокатора. Однако чтобы уметь восстанавливать состояние с файла на диске, нужно запоминать виртуальный адрес, по которому файл был отображён в предыдущем запуске программы, и при следующем запуске программы и чтении из файла сдвигать указатели в вершинах на разницу в виртуальных адресах отображений.

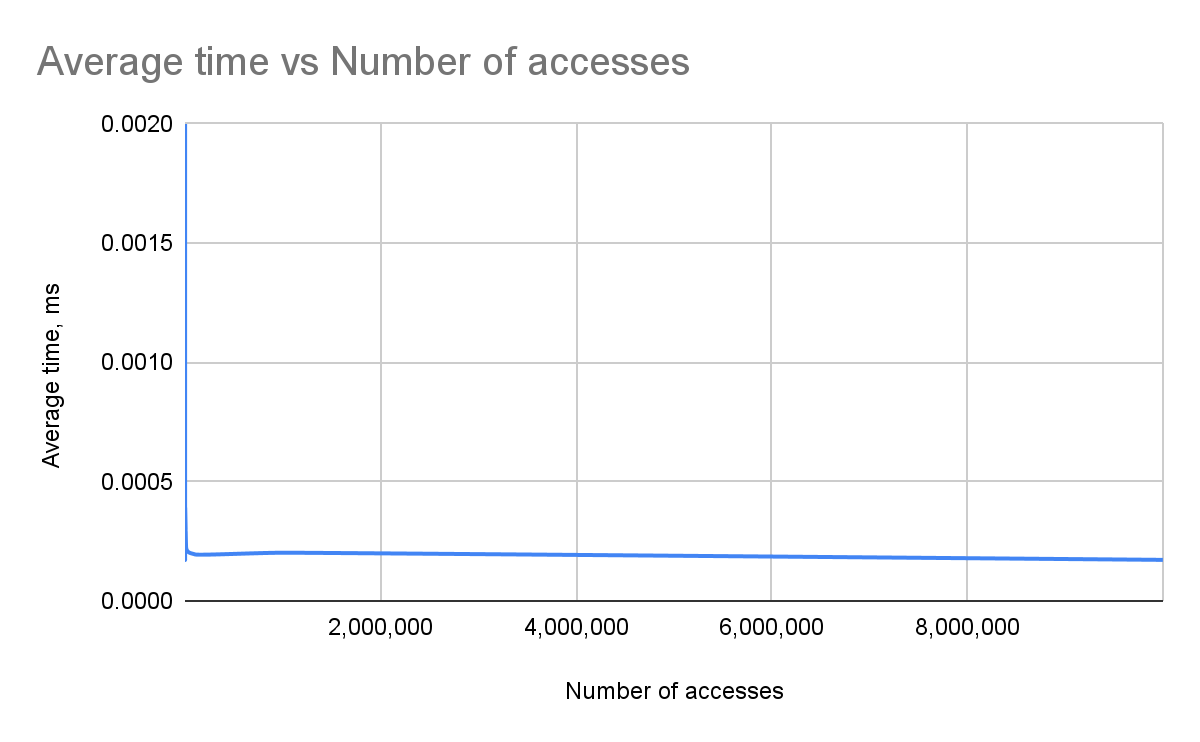
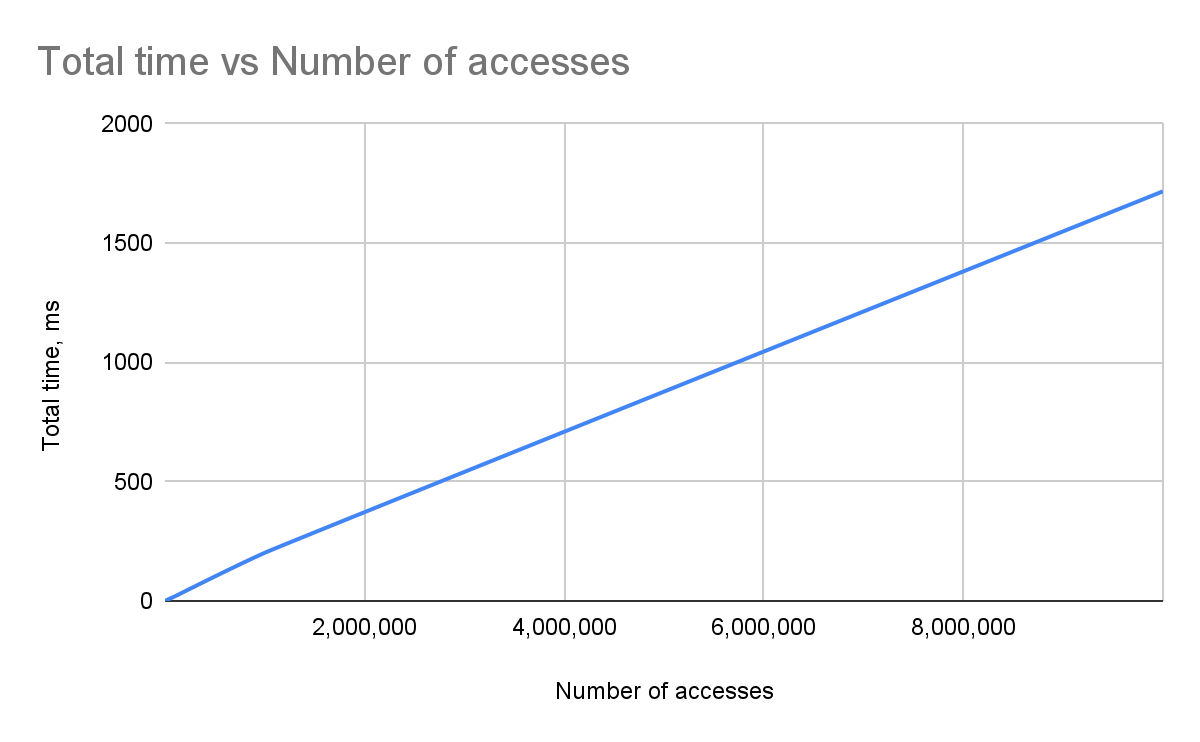
**Результаты тестов на производительность**

Ниже приведены результаты бенчмарков работы программы.

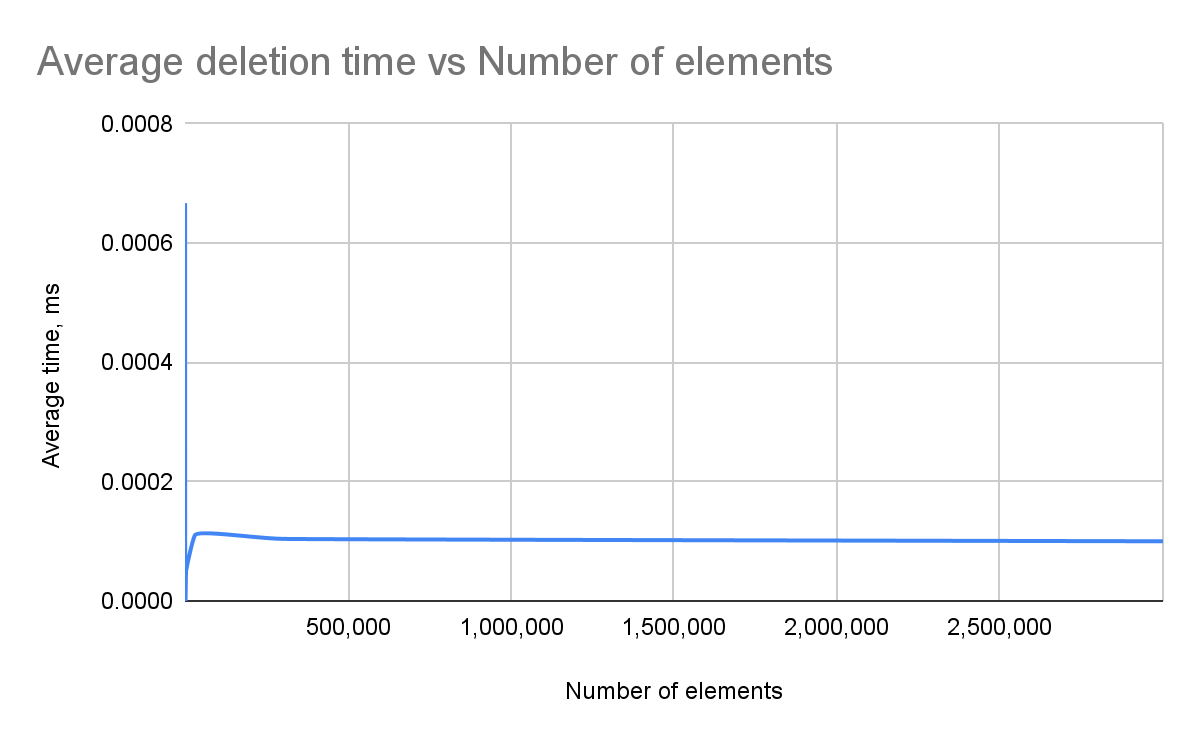
****







Как легко заметить из графиков выше, требуемая асимптотика достигается. Также можно показать, что время удаления (а также обновления/доступа) одного элемента дерева не зависит от количества представленных элементов в файле:



Примечание: пики, наблюдаемые на графиках при малом количестве добавлений/удалений/обновлений/доступов к элементам связаны с особенностями измерения времени работы программы (оверхедом).

**Вывод**: я реализовала модуль для эффективного хранения данных, представленных в виде документного древа. Я научилась работать с файлами, отображаемыми в память, и использовала эту технику в своём решении, что позволило работать с данными, суммарный размер которых превышает размер оперативной памяти. Также я написала тесты, проверяющие корректность работы программы, и тесты, измеряющие производительность различных операций над данными.