Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования
"Национальный Исследовательский Университет ИТМО"

Факультет Программной Инженерии и Компьютерных Технологий

Лабораторная работа №1

по дисциплине

«Низкоуровневое программирование»

Выполнил:
Студент группы P33302
Тюрин Святослав Вячеславович

Преподаватель
Кореньков Юрий Дмитриевич

Санкт Петербург
2023

Цель

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), поддержать 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/полями/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 $O(n*m) > t \rightarrow O(n+m)$, где n – количество представленных элементов данных обрабатываемого вида, m – количество фактически затронутых элементов данных

f. Показать, что размер файла данных всегда пропорционален 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

Описание работы

Система позволяющая хранить node в файле, а также производить операции над ними.

В системе есть следующие модули:

- Include – заголовочные файлы
- src – сорцы
  - main – входная точка
  - crud – реализация интерфейса операций БД
  - file – работа с самим файлом (открытие, чтение, запись, удаление)
  - test – обертка над операциями crud, для проверки производительности

Примеры работы программы

```
//////////////////// CREATE NODE ////////////////////////////
{
    id : 1
    vertex : 0
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
{
    id : 2
    vertex : 0
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
//////////////////// FIND NODE BY ID 1 ////////////////////////
{
    id : 1
    vertex : 0
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
```

```
//////////////////// CONNECT NODES 1 AND 2 ////////////////////////////
{
    id : 1
    vertex : 1
    neighbour 1, id : 2
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
{
    id : 2
    vertex : 1
    neighbour 1, id : 1
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
//////////////////// DELETE NODE BY ID 1 ////////////////////////////
{
    id : 2
    vertex : 0
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
//////////////////// FIND ALL ////////////////////////////
{
    id : 2
    vertex : 0
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
//////////////////// FIND BY FIELD - INTEGER VALUE - 10 ////////////////////
{
    id : 2
    vertex : 0
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
//////////////////// DELETE ALL ////////////////////////////
//////////////////// CREATE NODE ////////////////////////////
{
    id : 1
    vertex : 0
    integer : 10
    double : 1011.110
    boolean : 1
    string : new element
}
```

Аспекты реализации

В голове файла хранить структура header, которая содержит служебную информацию. В ней хранятся ссылки на первую node в файле и на последнюю, что дает нам двусвязный список, с помощью которого мы можем спокойно бегать по всем node’s в файле. Так же здесь, в двусвязном списке, хранятся hole’s, которые образуются если удалить не последний node в файле. Если в файле есть hole, то добавленный нами элемент встанет именно в эту hole. И встанет он без проблем, ибо все node одинакового размера.

```
// file's header
struct header {
    uint32_t signature;
    uint64_t first_hole_ptr; // link on the first hole in file
    uint64_t first_node_ptr; // link on the first node in file
    uint64_t last_node_ptr; // link on the last node in file
    uint64_t node_id; // current node id
} __attribute__((packed));

// file's hole
struct hole {
    uint64_t hole_ptr; // link on hole
    uint64_t size_of_hole; // hole size
    uint64_t prev_ptr;  // link on previous hole
    uint64_t next_ptr; // link on next hole
} __attribute__((packed));
```

Собственно, сама node и двусвязный список в ней. Проблема размерности node заключается в разной размерности string. Если в двух node строки разного размера, то заменить одну другой не получится, т.к. файл так не позволяет делать. Поэтому необходимо, чтобы node’s были одинакового размера. Это достигается благодаря ссылке внутри node на структуру string_save, которая в свою очередь хранит ссылку на string в файле.

```
// struct for string save in file
struct string_save {
    uint64_t size_of_string;
    uint64_t string_line_ptr;
} __attribute__((packed));

// main struct
struct node {
    uint64_t id; // node id
    uint64_t d; // vertex
    uint64_t nodes; // link on connected nodes
    uint64_t prev_ptr; // link on the previous node
    uint64_t next_ptr; // link on the next node
    uint64_t intgr; // integer
    uint64_t dbl; // double
    uint64_t bln; // boolean
    uint64_t strings_array_ptr; // link on struct "string_save" which have our string
} __attribute__((packed));
```

Операции:

- Добавление – мы записываем новую node в файл. В уже предпоследнюю обновляем ссылку на последнюю node, а в новую кладем ссылку на предыдущую.
- Поиск по id – мы бежим по всем node’s в файле и сравниваем их id с нужным и, как находим, достаем.
- Удаление по id – ищем нужную node в файле, удаляем, а затем смотрим есть ли у неё связанные node, если да, то удаляем и их. Немного манипуляций с перевязыванием ссылок в двусвязном списке и все готово.
- Соединение двух node’s по id – ищем эти две node’s в файле, читаем их. Затем в первую node кладем в связанную 2, а во вторую кладем в связанную 1.
- Обновление поля по id – ищем node по id, и заменяем значение нужного нам поля на новое.
- Поиск по значению поля – бежим по всему файлу и ищем node’s с нужным нам значением поля, кладем их в node_list, ибо их может быть и не одна штука.

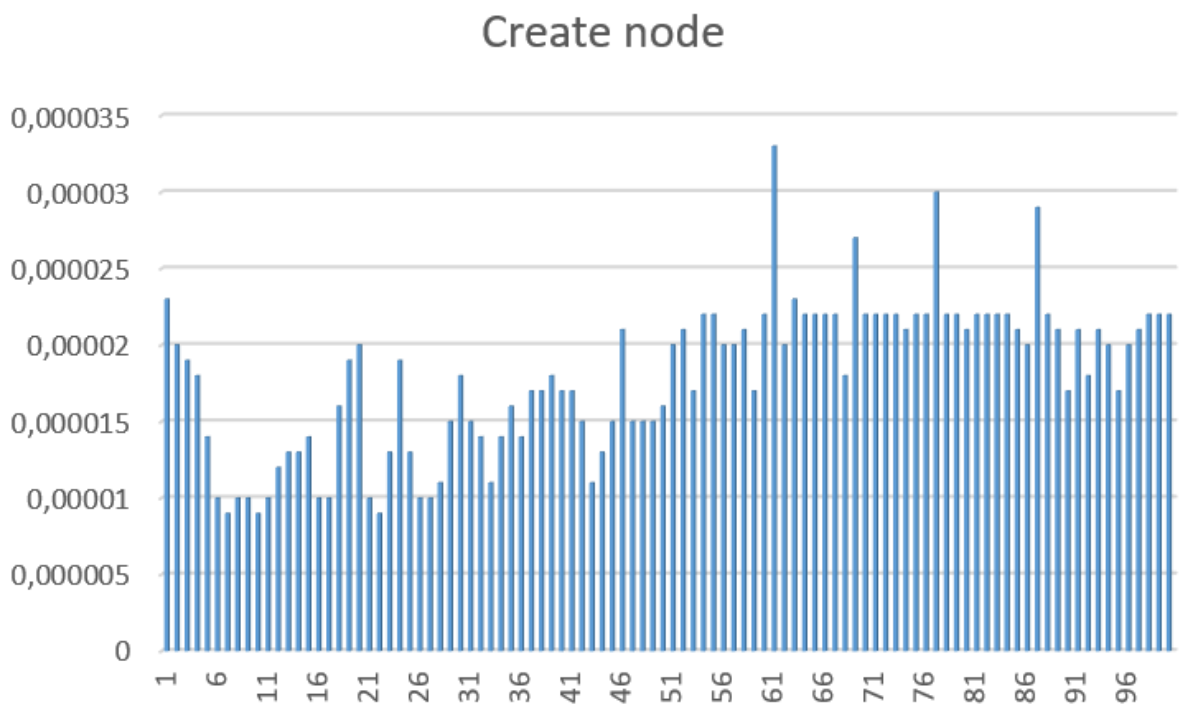
Результаты

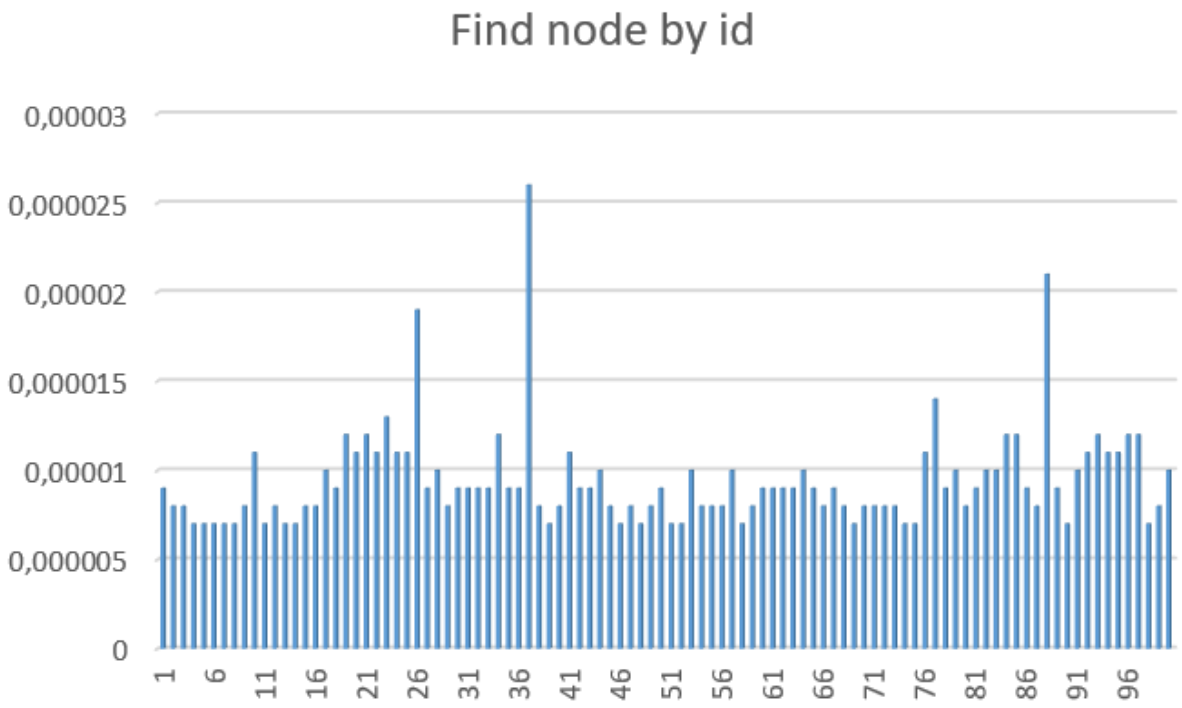
Для решения задача реализованы:

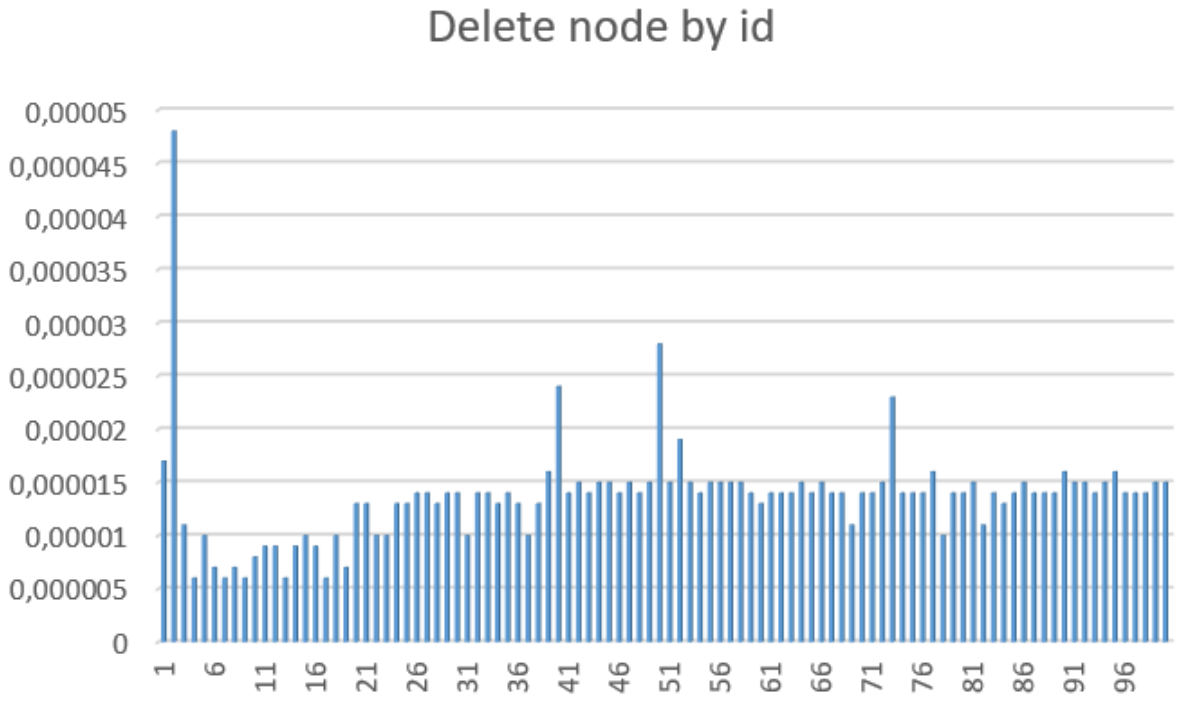
- Структуры header, node, hole
- Интерфейс для операций над данными в файле
- тесты, для проверки результата

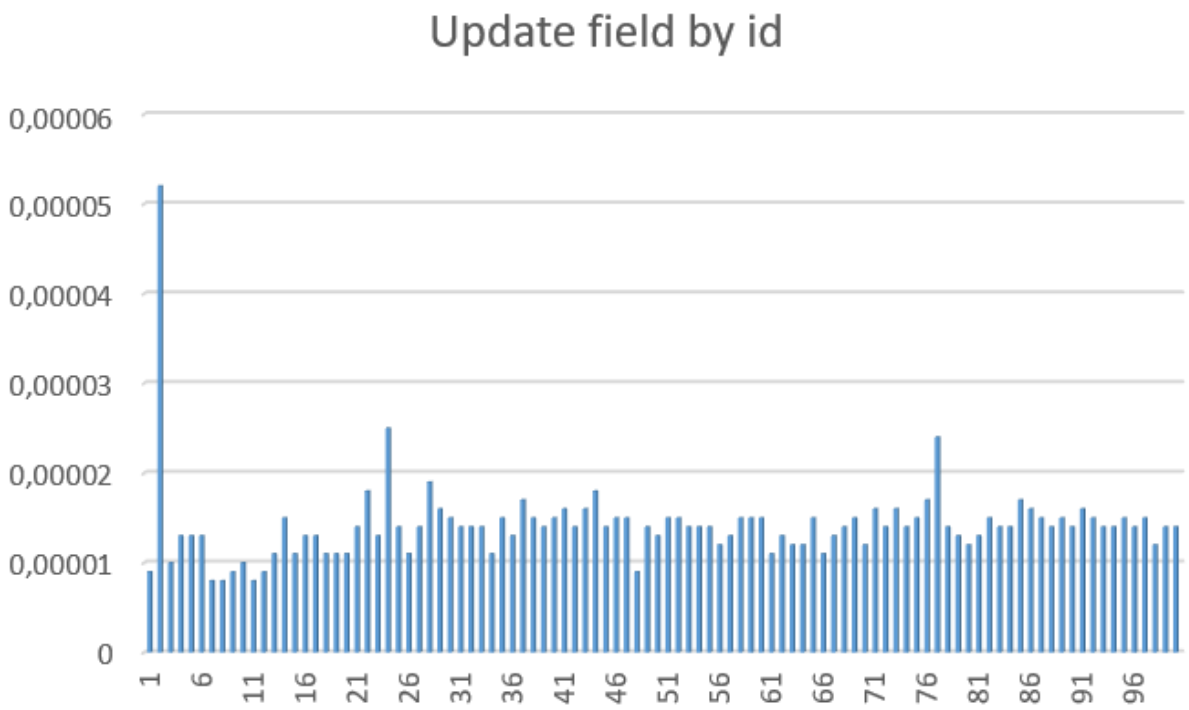
Семантика операций:

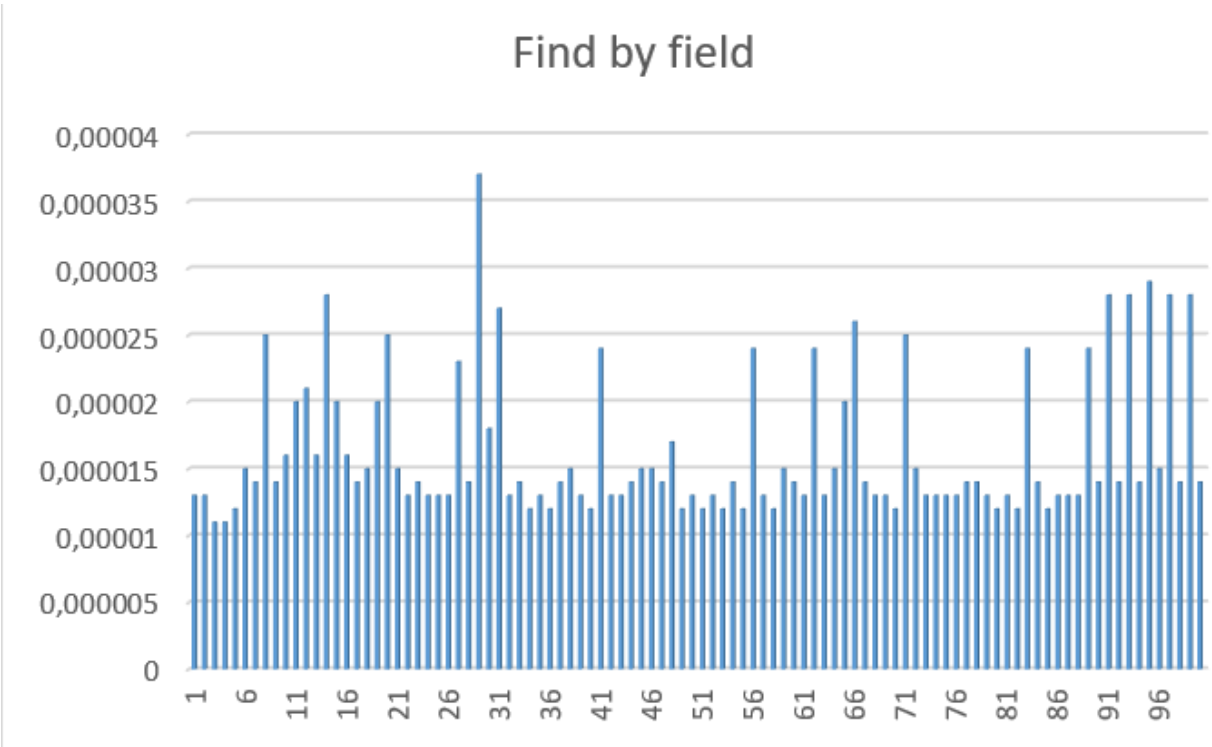
- Добавление O (1)
- Поиск по id O (n)
- Удаление по id O (n * m), где n весь файл, а m – фактически затронутые
- Обновление поля по id O (n)
- Поиск по значению поля O (n)

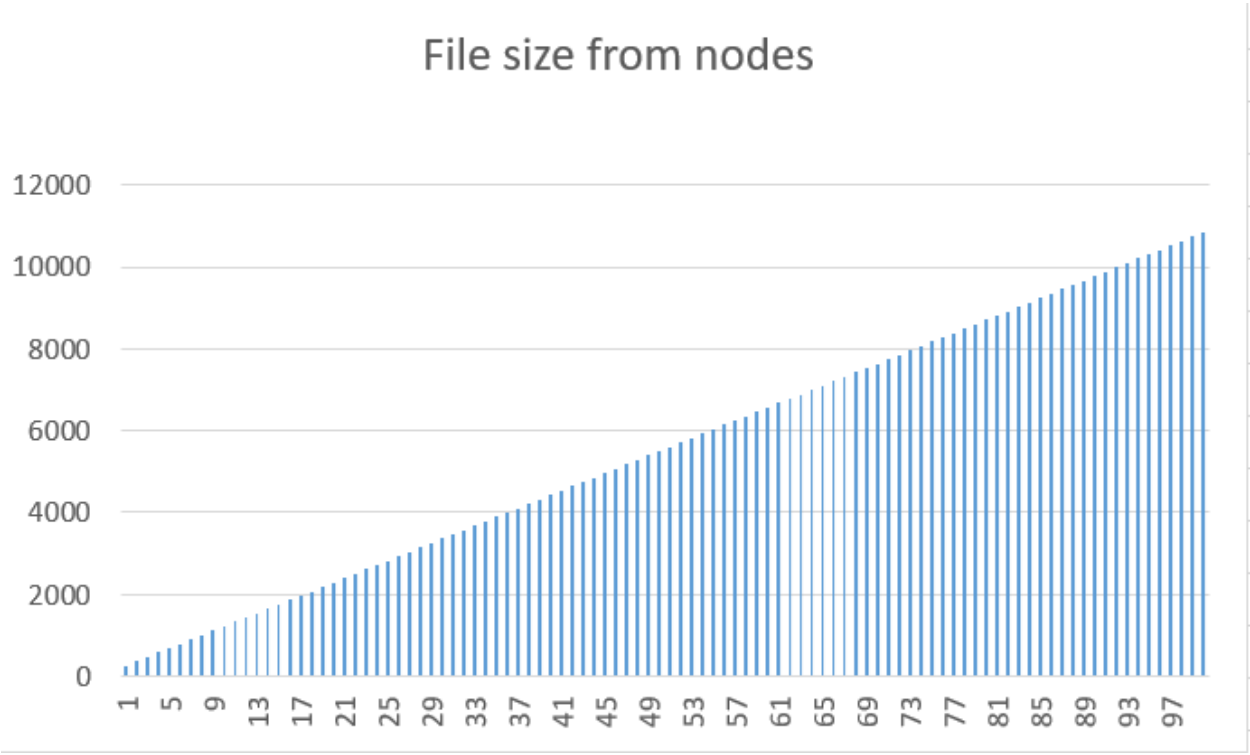



Find node by id
0,00003
0,000025
0,00002
0,000015
0,00001
0,000005
0
1
6
11
16
21
26
31
36
41
46
51
56
61
66
71
76
81
86
91
96

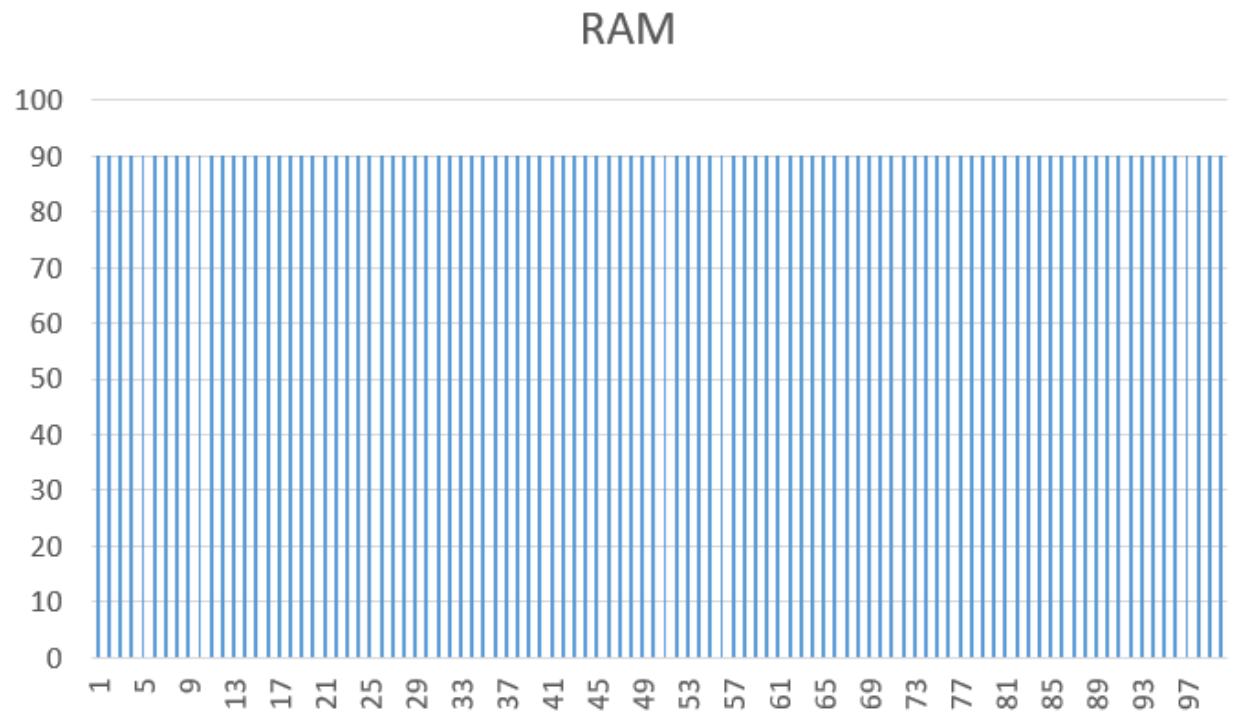

Delete node by id
0,00005
0,000045
0,00004
0,000035
0,00003
0,000025
0,00002
0,000015
0,00001
0,000005
0
1
6
11
16
21
26
31
36
41
46
51
56
61
66
71
76
81
86
91
96


Update field by id
0,00006
0,00005
0,00004
0,00003
0,00002
0,00001
0
1
6
11
16
21
26
31
36
41
46
51
56
61
66
71
76
81
86
91
96


Find by field
0,00004
0,000035
0,00003
0,000025
0,00002
0,000015
0,00001
0,000005
0
1
6
11
16
21
26
31
36
41
46
51
56
61
66
71
76
81
86
91
96



Утилита - gnome-system-monitor



Вывод: Тесты показали, что моя структура соответствует требованиям производительности при выполнение основных операций. При выполнении работы столкнулся с проблемой, что при выделение памяти для хранения строки – она занимает определённое место, и при попытки вставить новую ноду на то место с строкой другой длины происходит ошибка, поэтому было принято решение хранить в нодах ссылку на строки, а строки отдельно.