Университет ИТМО

Факультет программной инженерии и компьютерной техники
Направление подготовки 09.03.04 Информатика и вычислительная техника
Дисциплина «Низкоуровневое программирование»

Отчет

По лабораторной работе №1

Вариант 3 (Граф с атрибутами)

Выполнил:
Кузнецов Н. Д.
Преподаватель:
Кореньков Ю. Д.

Санкт-Петербург, 2023 г.

Цели:

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 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

Задачи:

Для выполнения поставленного ТЗ потребовалось придумать такую структуру бинарного файла, чтобы из него можно было извлекать/добавлять/изменять/удалять данные по требуемым сложностям алгоритмов (O(n)). После того, как это было выполнено, мною был написан модуль, который выполнял все вышенаписанные операции за необходимые сложности, и таким образом у меня получилась простая база данных, с настраиваемой схемой, а также со связями между ее узлами.

Описание работы:

Были созданы следующие структуры данных/перечисления:

1) Перечисление, которое показывает, какие типы данных могут храниться в нашей бд.

```
enum DataType { INT32, FLOAT, CHAR_PTR, BOOL };
```

2) Структура столбца бд (то, из каких полей будет состоять наша база данных):

```
struct Column {
        char name[MAX_NAME_LENGTH];
        enum DataType type;
```

```
        union {
            int32_t int32Value;
            float floatValue;
            char charValue[MAX_CHAR_VALUE];
            bool boolValue;
        };
};
```

3) Структура узла базы данных:

```
struct Node {
        struct Column* columns;
        int relationsCount;
        int relations[MAX_NODE_RELATIONS];
};
```

4) Структура самой базы данных:

```
struct GraphDB {
        int nodesCount;
        int columnsCount;
        struct Column* scheme;
        struct Node* nodes;
};
```

Реализация ответа на запрос пользователя к бд в моей реализации сделано через возвращаемое значений CRUD функций bool (true – операция прошла успешно, false – операция завершилась некорректно). Если это операция на получение значения, то в параметрах также необходимо передать указатель на запрошенное значение (например на Node). Далее в отчете CRUD функции будут указаны. Основные структуры и интерфейс бд лежит в файле graph.h.

Вместе с этим был написан простой рандомайзер значений каждого типа, поддерживаемого базой данных, реализованной в рамках данной лабораторной работы. Рандомайзер используется в файле main.c, в тестах, проверяющих сложности (O(n)) каждой из операций.

Аспекты реализации:

Бинарный файл, в котором хранится вся информация базы данных состоит из заголовка (в коде указывается, как header), а также из самих данных – записей базы данных.

Пример создания заголовка и сохранение его в бинарный файл:

```c
int main() {

    struct GraphDB db;
    db.nodesCount = 0;
    db.columnsCount = 0;
    db.nodes = NULL;
    db.scheme = NULL;

    int columnsCount = 4;

    struct Column scheme[columnsCount];
    scheme[0].type = INT32;
    strncpy(scheme[0].name, "Age\0", MAX_NAME_LENGTH);

    scheme[1].type = FLOAT;
    strncpy(scheme[1].name, "Balance\0", MAX_NAME_LENGTH);

    scheme[2].type = CHAR_PTR;
    strncpy(scheme[2].name, "Name\0", MAX_NAME_LENGTH);

    scheme[3].type = BOOL;
    strncpy(scheme[3].name, "isAdult\0", MAX_NAME_LENGTH);

    setScheme(&db, scheme, columnsCount);

    saveHeaderStructToFile(&db, "data.bin");

    return 0;
}
```

В заголовке хранится информация о схеме базы данных, количестве столбцов в схеме и количестве записей. Есть возможность настроить схему с разным количеством столбцов (при этом использовать можно только 4 типа данных). У каждого столбца есть свое название.

После создания заголовка и сохранения его в файл можно пользоваться API самой базы данных, который представлен в виде следующих функций:

```c
bool createNode(const char* fileName, char* inputString);
bool findNodeByIndex(const char* fileName, int index);
bool findNodes(const char* fileName, const char* columnName, const char* columnValue);
bool updateNodeByIndex(const char* fileName, const char* columnName, const char* columnValue, int index);
bool deleteNodeByIndex(const char* fileName, int index);
bool setNewRelation(const char* fileName, int index1, int index2);
bool clearAllRelationsOfNode(const char* fileName, int index);
```

примеры использования различных функций можно посмотреть в main.c файле

Результаты:

Для заполнения файла базы данных используем генератор случайных значений и обычный цикл, с указанием количества записей:

```c
// fill file with data
int main() {

    srand(time(NULL));

    int i;

    for(i = 0; i < 10000; i++) {

        int random_int_value = random_int(1, 100);
        float random_float_value = random_float(1.0, 10.0);
        char* random_string_value = random_string();
        bool random_bool_value = random_bool();

        char* db_string;
        asprintf(&db_string, "%d,%f,%s,%s", random_int_value, random_float_value, random_string_value, random_bool_value ? "true" : "false");

        bool result = createNode("data.bin", db_string);

        free(db_string);
    }

    return 0;
```

Для проверки количества записей и размера файла используем следующий скрипт:

```c
// Print header info and file size
int main() {

    struct GraphDB db;
    loadHeaderStructFromFile(&db, "data.bin");

    printf("File size: %d\n", getFileSize("data.bin"));

    printHeaderGraphDB(&db);

    return 0;
}
```

Пример вывода:

```
File size: 14527064
Column count: 4
Scheme:
        1. Age (INT32)
        2. Balance (FLOAT)
        3. Name (CHAR_PTR)
        4. isAdult (BOOL)
Number of nodes: 10004
```

***File size:** размер файла в байтах*

***Column count, scheme:** информация о схеме бд*

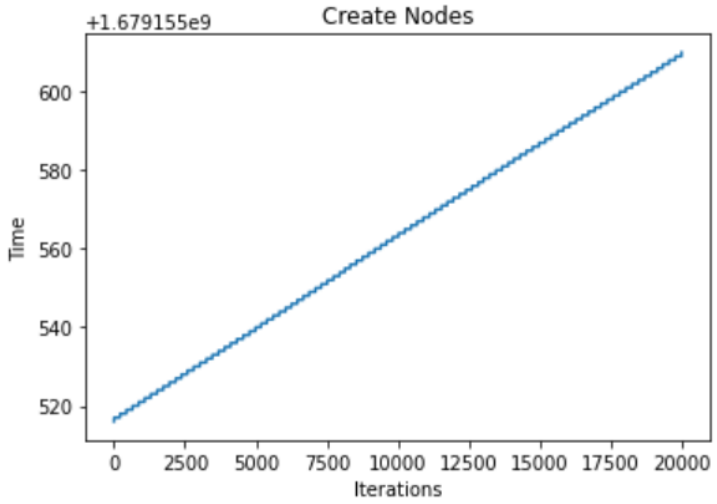
***Number of nodes:** количество вершин в нашей бд*

Тест №1 (проверяем сложность операции добавления вершин в графе):

Код:

```c
int main() {

    srand(time(NULL));

    FILE *fp;
    fp = fopen("timings.csv", "w+");

    int i;

    for(i = 0; i < 20000; i++) {

        time_t rawtime;
        time(&rawtime);

        fprintf(fp, "%d,%d\n", i, rawtime);

        int random_int_value = random_int(1, 100);
        float random_float_value = random_float(1.0, 10.0);
        char* random_string_value = random_string();
        bool random_bool_value = random_bool();

        char* db_string;
        asprintf(&db_string, "%d,%f,%s,%s", random_int_value, random_float_value, random_string_value, random_bool_value ? "true" : "false");

        bool result = createNode("data.bin", db_string);

        free(db_string);
    }

    return 0;
}
```

Результат (график):



Вывод:

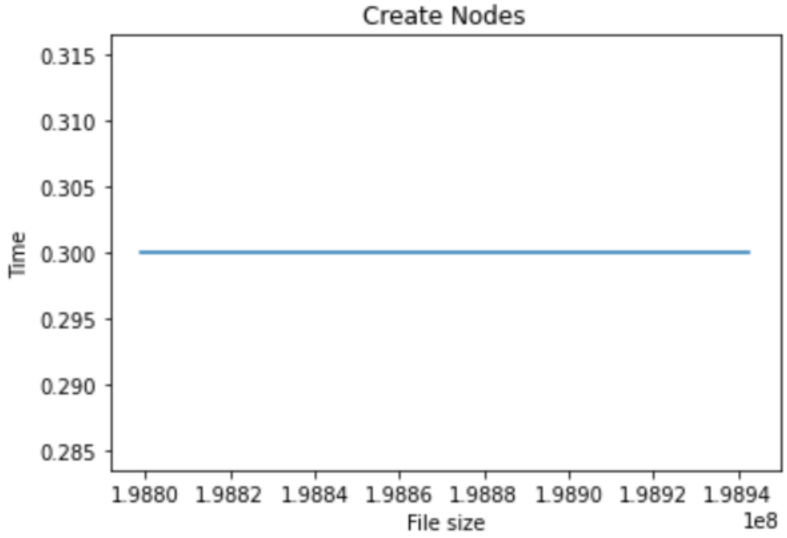
Т.к. по левой оси у нас время в Unix виде, то на графике отобразилась наклонная прямая, но также это говорит о том, что операция добавления выполняется за O(1). Если увеличить масштаб графика, то будет видно, что за некоторые промежутки времени выполняется несколько операций добавления (график выглядит в виде лесенки).

Тест №2 (проверяем сложность операции добавления вершины в граф в зависимости от размера файла):

Код:

```
int main() {

    srand(time(NULL));

    FILE *fp;
    fp = fopen("timings.csv", "w+");

    int i;

    for(i = 0; i < 100; i++) {

        time_t rawtime_start;
        time(&rawtime_start);

        int random_int_value = random_int(1, 100);
        float random_float_value = random_float(1.0, 10.0);
        char* random_string_value = random_string();
        bool random_bool_value = random_bool();

        char* db_string;
        asprintf(&db_string, "%d,%f,%s,%s", random_int_value, random_float_value, random_string_value, random_bool_value ? "true" : "false");

        bool result = createNode("data.bin", db_string);

        sleep(0.3);

        time_t rawtime_end;
        time(&rawtime_end);

        fprintf(fp, "%f,%d\n", rawtime_end - rawtime_start, getFileSize("data.bin"));

        free(db_string);
    }

    return 0;
```

Результат:



Вывод:

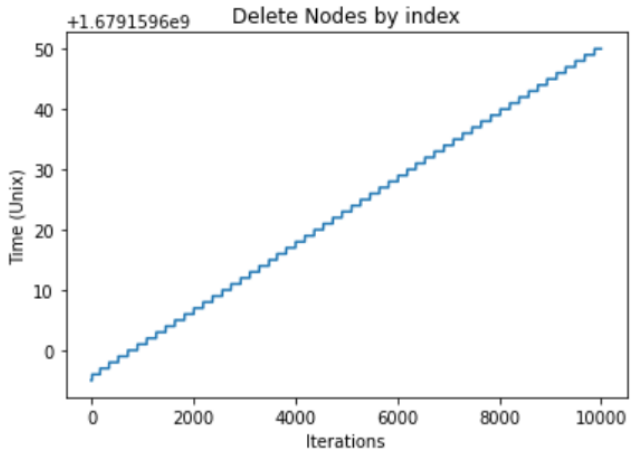
Таким образом, размер файла не влияет на скорость добавления вершины в граф.

Тест №3 (проверяем сложность операции удаления по индексу):

Код:

```c
int main() {

    srand(time(NULL));

    FILE *fp;
    fp = fopen("timings.csv", "w+");

    int i;

    for(i = 0; i < 10000; i++) {

        time_t rawtime;
        time(&rawtime);

        fprintf(fp, "%d,%d\n", i, rawtime);

        deleteNodeByIndex("data.bin", i);
    }

    return 0;
}
```

Результат:



Вывод:

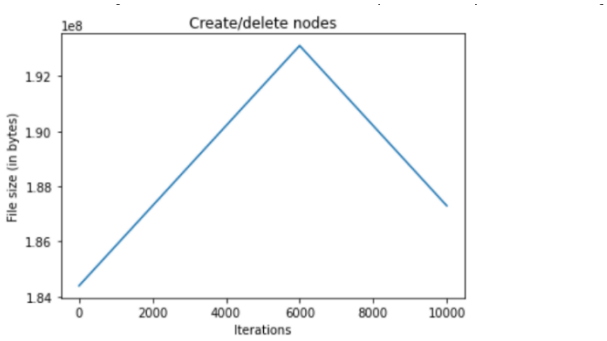
Вывод аналогичен выводу из теста №1 – операция удаления элемента по индексу выполняется за O(1), график выглядит в виде прямой, т.к. по вертикали у нас Unix время. Можно заметить, что график выглядит в виде лесенки, где в одну секунду времени выполняется какое-то количество удалений и так далее.

Тест №4 (проверяем размер файла от количества операций добавления/удаления):

Код:

```
int main() {

    srand(time(NULL));

    FILE *fp;
    fp = fopen("timings.csv", "w+");

    int i;

    for(i = 0; i < 10000; i++) {

        if (i < 6000) {
            int random_int_value = random_int(1, 100);
            float random_float_value = random_float(1.0, 10.0);
            char* random_string_value = random_string();
            bool random_bool_value = random_bool();

            char* db_string;
            asprintf(&db_string, "%d,%f,%s,%s", random_int_value, random_float_value, random_string_value, random_bool_value ? "true" : "false");

            bool result = createNode("data.bin", db_string);
        } else {
            deleteNodeByIndex("data.bin", i);
        }

        fprintf(fp, "%d,%d\n", i, getFileSize("data.bin"));
    }

    return 0;
}
```

Результат:



Вывод:

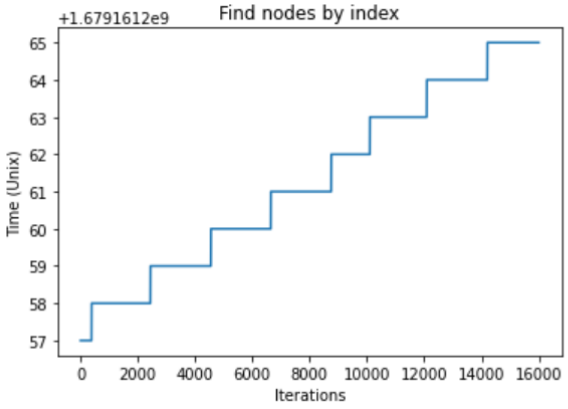
Размер файла изменяется пропорционально количеству выполненных с ним операций добавления и удаления записей.

Тест №5 (проверка операций поиска записей по индексу):

Код:

```
int main() {

    srand(time(NULL));

    FILE *fp;
    fp = fopen("timings.csv", "w+");

    int i;

    for(i = 0; i < 16000; i++) {

        findNodeByIndex("data.bin", i);

        time_t rawtime;
        time(&rawtime);

        fprintf(fp, "%d,%d\n", i, rawtime);
    }

    return 0;
}
```

Результат:



Вывод:

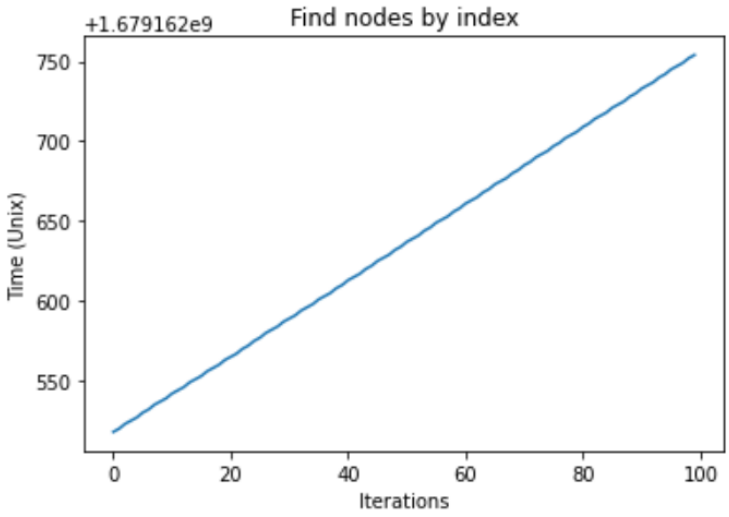
Вывод аналогичен выводам из тестов №1, 3 – операция поиска по индексу выполняется за O(1). В определенные моменты времени выполняется несколько итераций поиска по индексу.

Тест №6 (проверка операций поиска записей с условием):

Код:

```
int main() {

    srand(time(NULL));

    FILE *fp;
    fp = fopen("timings.csv", "w+");

    int i;

    for(i = 0; i < 100; i++) {

        findNodes("data.bin", "isAdult", "true");

        time_t rawtime;
        time(&rawtime);

        fprintf(fp, "%d,%d\n", i, rawtime);
    }

    return 0;
}
```

Результаты:



Вывод:

Сложность поиска узлов с условием напрямую зависит от количества записей в базе данных (сложность O(n)).

Вывод:

В рамках данной лабораторной работы были повторены и закреплены навыки работы в языке программирования C с бинарным файлом, с динамической памятью. Был реализован C модуль, представляющий из себя графовую базу данных, которая довольно производительна относительно своих CRUD операций.