

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РФ

Федеральное государственное автономное
образовательное учреждение высшего образования
«Национальный исследовательский университет ИТМО»

ФАКУЛЬТЕТ ПРОГРАММНОЙ ИНЖЕНЕРИИ И КОМПЬЮТЕРНОЙ ТЕХНИКИ

ЛАБОРАТОРНАЯ РАБОТА №1
по дисциплине
«Низкоуровневое программирование»

Вариант 9

Граф узлов с атрибутами

Студент:
Смирнова А. Ю.
Группа Р33301

Преподаватель:
Кореньков Юрий Дмитриевич

Санкт-Петербург, 2023

Contents

Задание	3
Реализация	4
Тестирование	10
Вывод	18

Задание

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 $O(n \cdot m) > t \rightarrow O(n+m)$, 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

Реализация

Задачи, которые я выделила для себя:

- Создать структуры данных на основе связного списка
- Реализовать управление файлом
- Реализовать базовые CRUD-операции
- Реализовать тестовую программу для демонстрации работоспособности решения

Структуры данных

```
typedef struct attribute_def { // структура для хранения описания атрибута
    char *name; // имя атрибута
    char type; // тип атрибута
    struct attribute_def *next; // указатель на следующий атрибут
} attribute_def;

enum attribute_def_type { // типы атрибутов
    ATTR_INT,
    ATTR_FLOAT,
    ATTR_BOOL,
    ATTR_STR
};

typedef struct relationship_def { // структура для хранения описания связей
    struct node_type_def *node_connect; // указатель на тип узла, к которому
    // осуществлена связь
    struct relationship_def *next; // указатель на следующую связь
} relationship_def;

typedef struct node_type_def { // структура для хранения описания типа узла
    struct node_type_def *next; // указатель на следующий тип узла
    char *name; // имя типа узла
    attribute_def *attribute_def_first;
    attribute_def *attribute_def_last;
    relationship_def *relationship_def_first; // указатель на первую связь
    relationship_def *relationship_def_last; // указатель на последнюю связь
    uint64_t file_root; // смещение от начала файла до списка узлов данного
    // типа
    uint64_t file_first_element; // смещение от начала файла до первого
    // элемента данного типа
    uint64_t file_last_element; // смещение от начала файла до последнего
    // элемента данного типа

    char *buffer; // буфер с данными
    int buffer_occupied_size; // размер занятой части буфера
    uint64_t adding; // флаг, указывающий, что в данный момент идет
    // добавление нового элемента
}
```

```

    uint64_t file_current_element; // смещение от начала файла до текущего
    элемента
    uint64_t file_previous_element; // смещение от начала файла до
    предыдущего элемента
} node_type_def;

typedef struct node { // структура для хранения самого узла
    node_type_def *node; // указатель на тип узла
    uint64_t file_previous_element; // смещение от начала файла до
    предыдущего элемента
    uint64_t file_current_element; // смещение от начала файла до текущего
    элемента
    struct node *next; // указатель на следующий узел
    struct node *prev; // указатель на предыдущий узел
} node;

typedef struct schema { // структура для хранения схемы базы данных
    node_type_def *first; // указатель на первый тип узла
    node_type_def *last; // указатель на последний тип узла
} schema;

typedef struct graph_db { // структура для хранения базы данных
    schema *schema; // указатель на схему
    char *filename;
    FILE *file;
    char *read_buf; // буфер для чтения
    uint64_t read_buf_pos; // позиция в буфере для чтения
    uint64_t read_buf_occupied_size; // размер занятой части буфера для
    чтения
    char *write_buf; // буфер для записи
    uint64_t write_buf_occupied_size; // размер занятой части буфера для
    записи
} graph_db;

attribute_def *create_attribute_def(node_type_def *node_type, char *name,
char type); // создание описания атрибута

void delete_attribute_def(attribute_def *attr); // удаление описания атрибута

relationship_def *create_relationship_def(node_type_def *connect_from,
node_type_def *connect_to); // создание описания связи

void delete_relationship_def(relationship_def *rel); // удаление описания
связи

node_type_def *create_node_type_def(schema *schema, char *name); // создание
описания типа узла

void delete_node_type_def(node_type_def *node_type); // удаление описания
типа узла

void create_node(graph_db *db, node_type_def *node_type); // создание узла

```

```

int delete_node(graph_db *db, node_type_def *node_type); // удаление узла

void post_node(graph_db *db, node_type_def *node_type); // добавление узла в
базу данных

int get_node(graph_db *db, node_type_def *node_type); // получение узла из
базы данных

schema *create_schema(); // создание схемы

void delete_schema(schema *schema); // удаление схемы

graph_db *create_graph_db(schema *schema, char *filename); // создание базы
данных

void delete_graph_db(graph_db *db); // удаление базы данных

bool next_node(graph_db *db, node_type_def *node_type); // переход к
следующему узлу

void restart_node_pointer(graph_db *db, node_type_def *node_type); // переход
к первому узлу

void set_value_for_attribute_of_node(graph_db *db, node_type_def *node_type,
char *attr_name, float value); // установка значения атрибута узла

float get_attribute_value_of_node(node_type_def *node_type, char *attr_name);
// получение значения атрибута узла

attribute_def *search_attribute_def(node_type_def *node, char *name, int
*num); // поиск описания атрибута

```

Представление структур в виде Linked list решают основную проблему медленной работы с записью, поиском, удалением узлов. Операции вставки и удаления узлов выполняются эффективно, так как требуют только изменения ссылок на узлы. Операция поиска может быть менее эффективной, так как для поиска элемента нужно просмотреть все узлы.

Базовая работа с файлом

```

#define BUFFER_SIZE (1024 * 32) // фиксированный размер буфера для операций
чтения и записи

extern size_t memory_used;

enum file_record_type {
    REC_EMPTY,
    REC_STRING, // для удобства хранения строк произвольной длины
    REC_NODE
};

void free_memory_counter(size_t size);

```

```

void *allocate_memory(size_t size);

size_t get_memory_used();

void db_fflush(graph_db *db);

void db_fwrite(void *buf, uint64_t item_size, int n_items, graph_db *db);

void db_fread(void *buf, int item_size, int n_items, graph_db *db);

void db_fseek(graph_db *db, long int offset, int whence);

long int db_ftell(graph_db *db);

void db_fclose(graph_db *db);

void write_buffer(char *buffer, int *n_buffer, float what);

```

Для хранения строк было решено использовать следующий способ

```

uint64_t string_init(graph_db *db, char *s) {
    unsigned char Type = REC_STRING; // обозначение типа записи
    uint64_t length = strlen(s); // длина строки
    uint64_t n = sizeof(uint64_t) + sizeof(unsigned char) + 1 + length; //
размер записи, включаем туда длину строки, тип записи и нулевой символ
    uint64_t result; // смещение от начала файла до записи строки
    db_fseek(db, 0, SEEK_END); // переходим в конец файла
    result = db_ftell(db); // запоминаем смещение от начала файла до конца
    db_fwrite(&n, sizeof(n), 1, db); // записываем размер записи
    db_fwrite(&Type, sizeof(Type), 1, db); // записываем тип записи
    db_fwrite(s, length + 1, 1, db); // записываем строку
    db_fflush(db);
    return result;
}

```

Эта функция предназначена для извлечения строки по ее смещению. Строка создается в динамической памяти.

```

char *string_get(graph_db *db, uint64_t offset) {
    unsigned char type;
    char *result;
    uint64_t length; // длина строки
    uint64_t n; // размер записи
    db_fseek(db, offset, SEEK_SET); // переходим по указанному смещению
    db_fread(&n, sizeof(n), 1, db); // считываем размер записи
    db_fread(&type, sizeof(type), 1, db);
    if (type != REC_STRING)
        return NULL;
    length = n - sizeof(int) - sizeof(unsigned char); // вычисляем длину
строки
    result = allocate_memory(length); // выделяем память под строку
    db_fread(result, length, 1, db); // считываем строку
    return result;
}

```

Создание графа узлов с атрибутами осуществляется через сохранение данных и метаданных, необходимых для правильной интерпретации и работы с фактическими данными.

Выборка осуществляется через итерацию и сравнение.

Удаление осуществляется через переназначение ссылок и освобождение ресурсов.

Для проверки утечек памяти используется переменная `memory_used` для отслеживания количества использованной памяти в результате каждой микро- и макрооперации.

```
void delete_attribute_def(attribute_def *attr) {
    memory_used -= strlen(attr->name) + 1;
    free(attr->name);
    memory_used -= sizeof(attribute_def);
    free(attr);
}
```

Она помогла мне обнаружить колоссальные утечки памяти, которые я не замечала в начале, думая, что все у меня хорошо.

Взаимодействие

Файл `query.h` содержит интерфейс взаимодействия – набор функций и структур данных для создания запросов. Основные структуры данных включают перечисления для типов условий и операндов, структуры для представления условий и операндов.

```
enum condition_type {
    COND_LESS,
    COND_GREATER,
    COND_EQUAL,
    COND_NOT_EQUAL,
    COND_AND,
    COND_OR,
    COND_NOT
};

enum operand_type {
    OP_NUMERIC,
    OP_STR,
    OP_NAME,
    OP_COND
};

typedef struct {
    unsigned char type;
    union {
        struct condition *op_cond;
        char *op_str;
        float op_numeric;
    };
};
```



```

        char *op_name;
    };
} operand_type;

typedef struct condition {
    unsigned char type;
    operand_type *op1;
    operand_type *op2;
} condition;

```

```

uint64_t string_init(graph_db *db, char *str);

char *string_get(graph_db *db, uint64_t offset);

condition *create_condition_numeric(unsigned char operation, char *name,
float value);

condition *create_condition_string(unsigned char operation, char *name, char
*value);

condition *create_condition_condition(unsigned char operation, condition
*cond1, condition *cond2);

void delete_condition(condition *cond);

node *select_query(graph_db *db, uint32_t n_links, ...);

void free_node_set(graph_db *db, node *set);

void get_node_from_set(graph_db *db, node *set);

void delete_query(graph_db *db, uint32_t n_links, ...);

void update_query(graph_db *db, char *attr_name, float value, uint32_t
n_links, ...);

void free_operand(operand_type *op);

int test_node_condition(graph_db *db, node_type_def *node, condition *cond);

node *query_all_nodes_of_type(graph_db *db, node_type_def *node, condition
*cond);

```

Тестирование

Будем вставлять строки рандомной длины.

```
#define INSERTION_TOTAL 400000
#define INSERTION_CHECKPOINT 2000

void test_insertion() {
    fprintf(csv_file, "Iteration,Time,MemoryUsed\n");
    printf("-----\nINSERTION\n-----\n");
    schema *sch = create_schema();
    node_type_def *node1 = create_node_type_def(sch, "type_1");
    create_attribute_def(node1, "attribute", ATTR_STR);
    graph_db *db = create_graph_db(sch, "database.db");

    get_time();
    for (int i = 0; i < INSERTION_TOTAL; i++) {
        create_node(db, node1);
        char *str = malloc(sizeof(char) * (rand() % 1000 + 1));
        set_value_for_attribute_of_node(db, node1, "attribute",
string_init(db, str));
        post_node(db, node1);

        if ((i + 1) % INSERTION_CHECKPOINT == 0) {
            fprintf(csv_file, "Items: %6d; Time: %9.6f; Memory: %6zu\n", i +
1, get_time(), get_memory_used());
            printf("Items: %6d; Time: %9.6f; Memory: %6zu\n", i + 1,
get_time(), get_memory_used());
        }
    }
    fclose(csv_file);
    delete_graph_db(db);
    memory_leak_check();
}
```

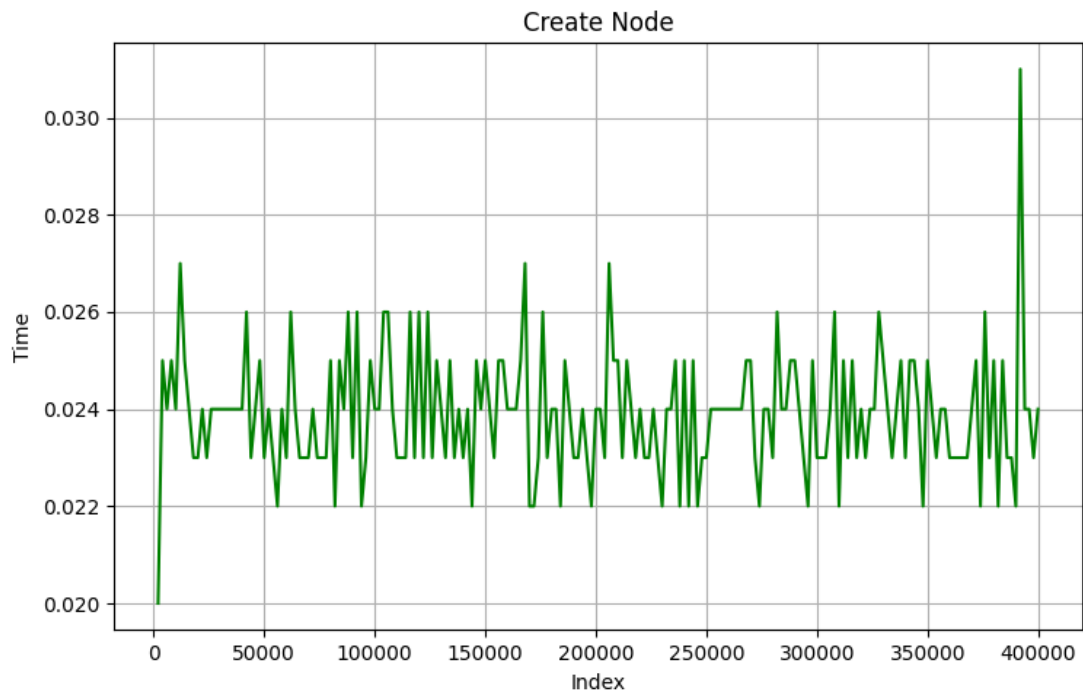


Рис. 1. Вставка за $O(1)$

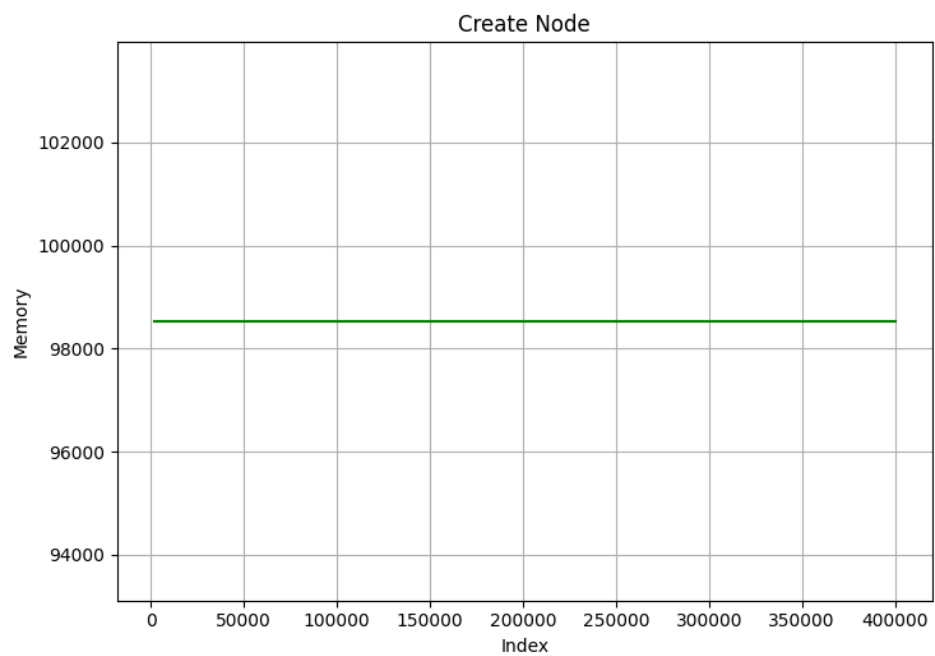


Рис.2. Потребление памяти во время вставки статично

```

#define DELETION_TOTAL 40000
#define DELETION_CHECKPOINT 200

void test_deletion() {
    printf("-----\ndeletion\n-----\n");
    schema *sch = create_schema();
    node_type_def *node1 = create_node_type_def(sch, "type_1");
    create_attribute_def(node1, "attribute", ATTR_STR);
    graph_db *db = create_graph_db(sch, "database.db");

    for (int i = 0; i < DELETION_TOTAL; i += DELETION_CHECKPOINT) {
        for (int i2 = 0; i2 < i; i2++) {
            create_node(db, node1);
            char *str = malloc(sizeof(char) * (rand() % 1000 + 1));
            set_value_for_attribute_of_node(db, node1, "attribute",
string_init(db, str));
            post_node(db, node1);
        }
        get_time();
        delete_query(db, 1, node1, NULL);

        fprintf(csv_file, "Items: %6d; Time: %9.6f; Memory: %6zu\n", i,
get_time(), get_memory_used());
        printf("Items: %6d; Time: %9.6f; Memory: %6zu\n", i, get_time(),
get_memory_used());
    }
    delete_graph_db(db);
    memory_leak_check();
}

```

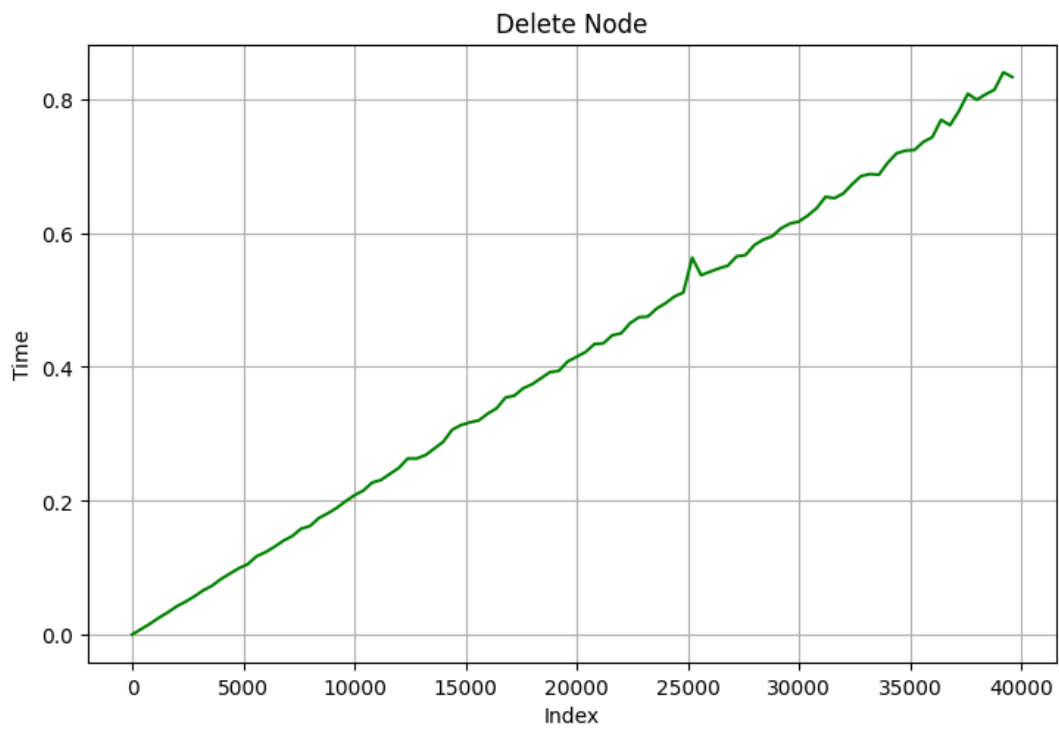


Рис. 3. Удаление за $O(n)$

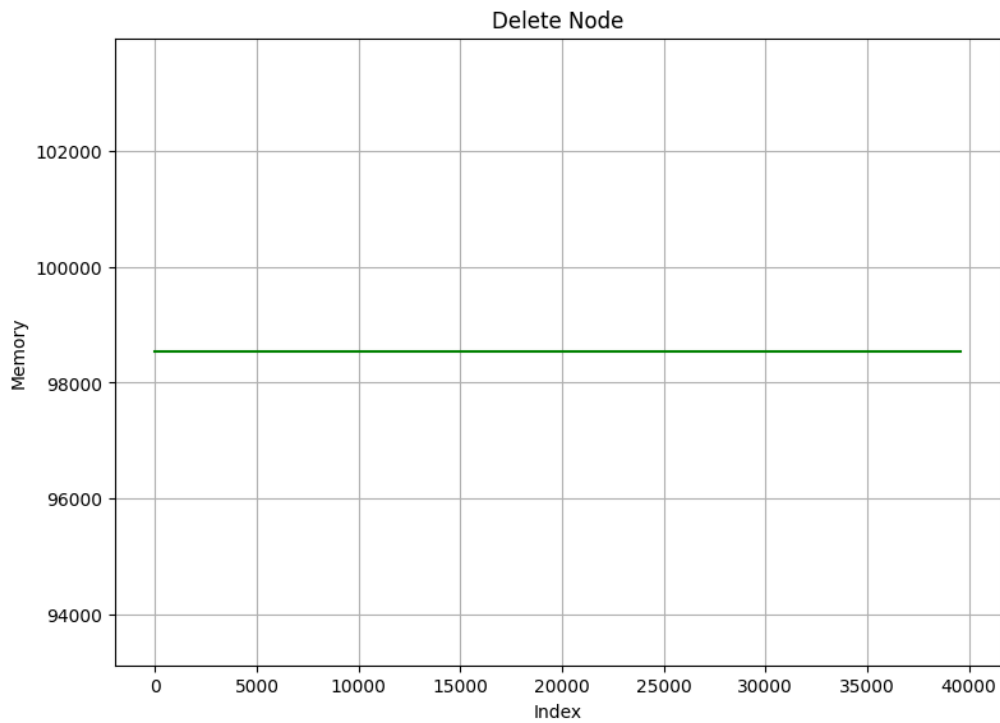


Рис. 4. Потребление памяти во время удаления статично

```

#define SELECTION_TOTAL 40000
#define SELECTION_CHECKPOINT 200

void test_selection() {
    fprintf(csv_file, "Iteration,Time,MemoryUsed\n");
    printf("-----\nSELECTION\n-----\n");
    schema *sch = create_schema();
    node_type_def *node1 = create_node_type_def(sch, "type_1");
    create_attribute_def(node1, "attr", ATTR_INT);
    graph_db *db = create_graph_db(sch, "database.db");

    condition *cond = create_condition_numeric(COND_LESS, "attr", 10);

    for (int i = 0; i < SELECTION_TOTAL; i += SELECTION_CHECKPOINT) {
        for (int i2 = 0; i2 < i; i2++) {
            create_node(db, node1);
            set_value_for_attribute_of_node(db, node1, "attr", 5);
            post_node(db, node1);
        }
        get_time();
        node *set = select_query(db, 1, node1, cond);
        printf("Items: %6d; Time: %9.6f; Memory: %6zu\n", i, get_time(),
get_memory_used());
        fprintf(csv_file, "Items: %6d; Time: %9.6f; Memory: %6zu\n", i,
get_time(), get_memory_used());
        free_node_set(db, set);
        delete_query(db, 1, node1, NULL);
    }

    delete_condition(cond);
    delete_graph_db(db);
    memory_leak_check();
}

```

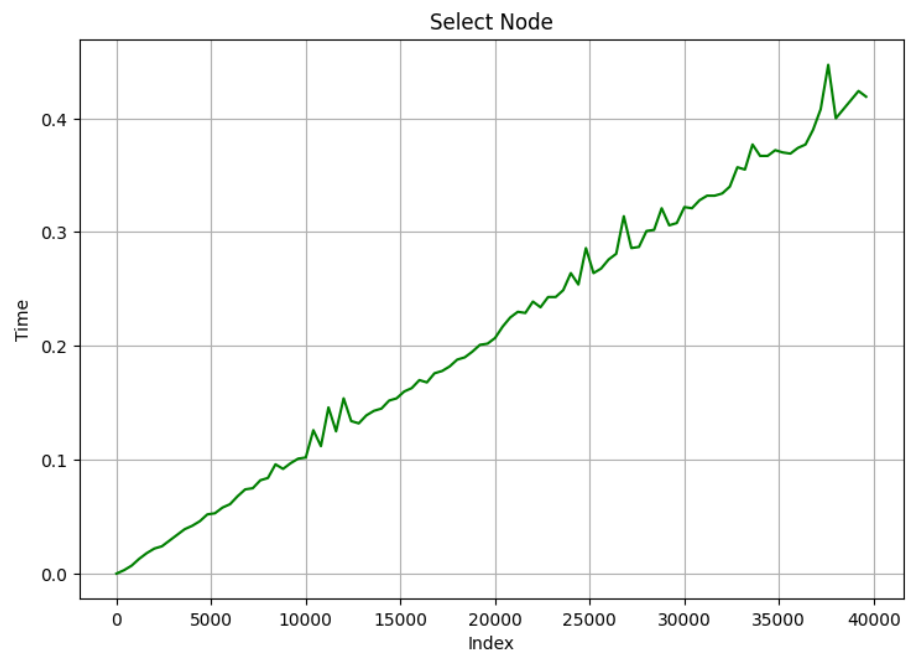


Рис. 5. Выборка за $O(n)$

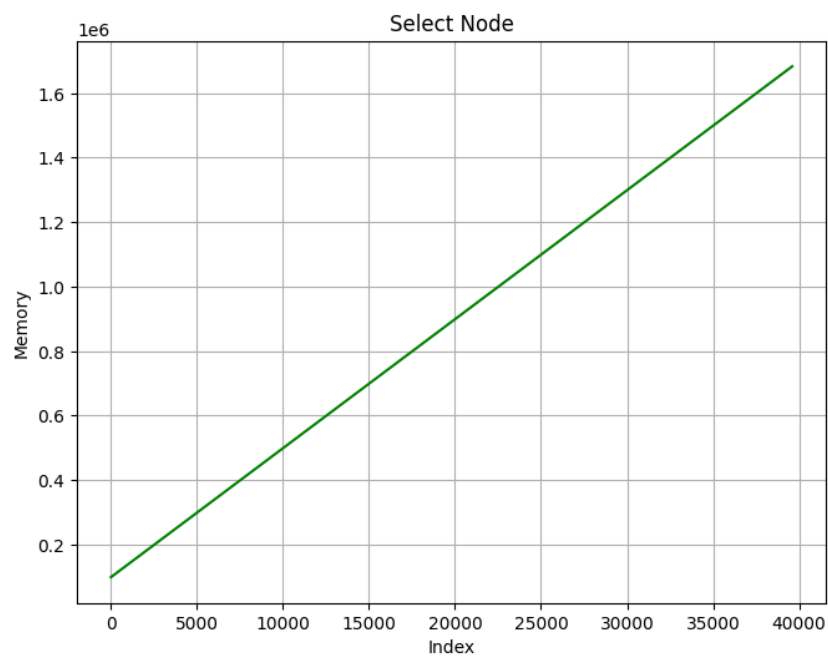


Рис. 6. Потребление памяти во время выборки пропорционально количеству получаемых данных

```

#define UPDATE_TOTAL 40000
#define UPDATE_CHECKPOINT 200

void test_update() {
    printf("-----\nUPDATE\n-----\n");
    schema *sch = create_schema();
    node_type_def *node1 = create_node_type_def(sch, "type_1");
    create_attribute_def(node1, "attribute", ATTR_STR);
    graph_db *db = create_graph_db(sch, "database.db");

    for (int i = 0; i < UPDATE_TOTAL; i += UPDATE_CHECKPOINT) {
        for (int i2 = 0; i2 < i; i2++) {
            create_node(db, node1);
            set_value_for_attribute_of_node(db, node1, "attribute",
string_init(db, "Hello world"));
            post_node(db, node1);
        }
        get_time();
        update_query(db, "attribute", string_init(db, "Goodbye planet"), 1,
node1, NULL);
        fprintf(csv_file, "Items: %6d; Time: %9.6f; Memory: %6zu\n", i,
get_time(), get_memory_used());
        printf("Items: %6d; Time: %9.6f; Memory: %6zu\n", i, get_time(),
get_memory_used());
        delete_query(db, 1, node1, NULL);
    }

    fclose(csv_file);
    delete_graph_db(db);
    memory_leak_check();
}

```

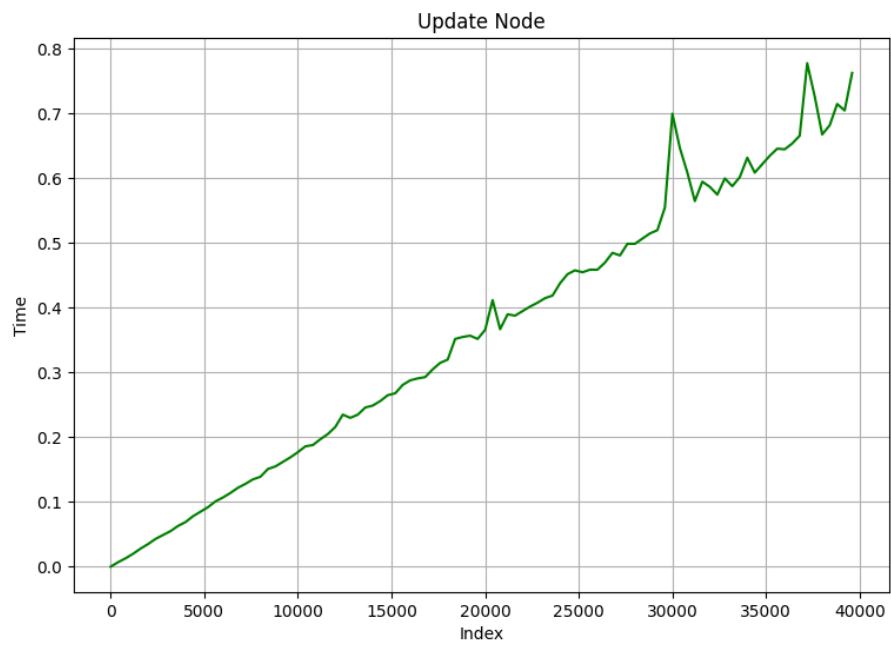



Рис. 7. Обновление за $O(n)$

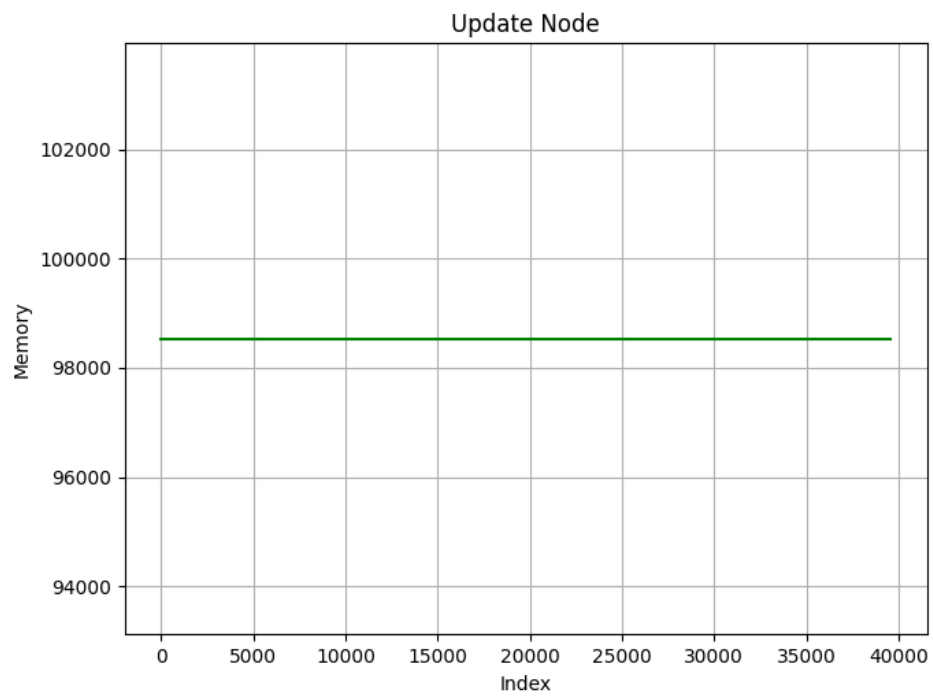


Рис. 8. Потребление памяти во время обновления статично

Вывод

Эта лабораторная научила меня работать со структурами, управлять памятью, проектировать непростые решения, работать с данными и метаданными. Это было сложно, но, оказывается, реально.