МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ образовательное учреждениевысшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Кафедра защиты информации



**ОТЧЁТ по ПРАКТИЧЕСКОЙ работе №2**

**по дисциплине «ПРОГРАММИРОВАНИЕ»**

**«Разработка сетевого интерфейса системы управления базой данных»**

|  |  |
| --- | --- |
| Факультет: АВТФ  Группа: АБс-222  Студент: Линкер Э.С. | Преподаватель: Архипова А.Б. |

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |

**Цели и задачи работы**: Изучение тем: Сеть, потоки, процессы и асинхронность.

**Задание к работе**:

Требуется реализовать сетевой интерфейс для СУБД из практики 1.

При запуске приложения из первой практики СУБД должна

ожидать соединение по протоколу tcp на порту 6379. При

подключении требуется обработать запрос либо в отдельном

потоке, либо в отдельном процессе, либо асинхронной задачей

и отдать результат в ответ на запрос.

Также необходимо позаботится о блокировках на структуре БД

если используются потоки или процессы.

**Текст программы**

**Сервер:**

// Отключаем предупреждения о небезопасных функциях

#define \_CRT\_SECURE\_NO\_WARNINGS

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 1000

#include <winsock2.h>

#include <ws2tcpip.h>

#include <windows.h>

#pragma comment(lib, "ws2\_32.lib")

#define PORT 6379

#define BACKLOG 10

#define BUFFER\_SIZE 104857600

HANDLE mutex;

int need\_unic\_key = 1;

// Структура для представления стека

typedef struct {

char\*\* elements; // Массив указателей на элементы стека

int top; // Индекс вершины стека

} Stack;

// Функция для инициализации стека

Stack\* InitStack() {

Stack\* stack = (Stack\*)malloc(sizeof(Stack)); // Выделяем память под структуру стека

stack->elements = (char\*\*)malloc(MAX\_SIZE \* sizeof(char\*)); // Выделяем память под массив указателей на элементы стека

stack->top = -1; // Устанавливаем индекс вершины стека в -1, что означает пустой стек

return stack; // Возвращаем указатель на стек

}

// Функция для добавления элемента в стек

void SPUSH(Stack\* stack, char\* element) {

if (stack->top >= MAX\_SIZE - 1) { // Проверяем, не переполнен ли стек

printf("Stack full\n"); // Выводим сообщение об ошибке

return; // Завершаем функцию

}

stack->top++; // Увеличиваем индекс вершины стека на 1

stack->elements[stack->top] = \_strdup(element); // Копируем элемент в массив указателей на элементы стека по индексу вершины

}

// Функция для удаления элемента из стека

void SPOP(Stack\* stack) {

if (stack->top < 0) { // Проверяем, не пустой ли стек

printf("Stack is empty\n"); // Выводим сообщение об ошибке

return NULL; // Возвращаем NULL

}

printf("item or basename:%s\n", stack->elements[stack->top]); // Выводим элемент на вершине стека

char\* element = stack->elements[stack->top]; // Сохраняем указатель на элемент на вершине стека

stack->top--; // Уменьшаем индекс вершины стека на 1

}

// Функция для выполнения операций со стеком в зависимости от запроса

Stack\* ExeStack(Stack\* other, const char\* filename, const char\* basename, const char\* item, const char\* query) {

FILE\* file = fopen(filename, "r"); // Открываем файл для чтения

if (file == NULL) { // Проверяем, не произошла ли ошибка при открытии файла

printf("Error opening file\n"); // Выводим сообщение об ошибке

return NULL; // Возвращаем NULL

}

Stack\* stack = InitStack(); // Инициализируем стек

char line[MAX\_SIZE][MAX\_SIZE] = { 0 }; // Объявляем двумерный массив символов для хранения строк из файла

char c; char tempbase[MAX\_SIZE] = { 0 }; // Объявляем переменные для хранения текущего символа и текущего имени базы

int sch\_st = 0; int nal\_st = 0; int bel = 1; // Объявляем переменные для хранения счетчика добавленных элементов, счетчика найденных баз и флага принадлежности к стеку

for (int i = 0; i + sch\_st < MAX\_SIZE - 3; i++) { // Цикл по строкам файла

fscanf(file, "%s", line[i]); // Считываем строку из файла

c = line[i][0]; // Сохраняем первый символ строки

if (c == '#') { // Если первый символ строки - решетка, то это означает начало новой базы для стека

bel = 1; // Устанавливаем флаг принадлежности к стеку в 1

if ((basename != NULL) && (strcmp(tempbase, basename) == 0) && (strcmp(query, "SPUSH") == 0)) { // Если задано имя базы, и оно совпадает с текущим именем базы, и запрос - добавление элемента в стек

SPUSH(stack, item); // Добавляем элемент в стек

nal\_st += 1; // Увеличиваем счетчик найденных баз на 1

sch\_st += 1; // Увеличиваем счетчик добавленных элементов на 1

}

for (int v = 0; v < MAX\_SIZE; v++) { // Цикл по символам текущего имени базы

tempbase[v] = 0; // Обнуляем символы текущего имени базы

}

for (int j = 1; line[i][j] != '\0'; j++) { // Цикл по символам строки, начиная со второго

tempbase[j - 1] = line[i][j]; // Копируем символы строки в текущее имя базы, пропуская решетку

}

}

if (c == '%' || c == '$' || c == '^') { // Если первый символ строки - процент, доллар или крышка, то это означает начало новой базы для очереди, множества или хеш-таблицы соответственно

bel = 0; // Устанавливаем флаг принадлежности к стеку в 0

}

if (c == '\n') { // Если первый символ строки - перенос строки, то это означает пустую строку

continue; // Продолжаем цикл

}

if (bel == 1) { // Если флаг принадлежности к стеку равен 1

SPUSH(stack, line[i]); // Добавляем строку в стек

}

if (bel == 0) { // Если флаг принадлежности к стеку равен 0

SPUSH(other, line[i]); // Добавляем строку в другой стек

}

if (feof(file)) { // Если достигнут конец файла

if ((basename != NULL) && (strcmp(tempbase, basename) == 0) && (strcmp(query, "SPUSH") == 0)) { // Если задано имя базы, и оно совпадает с текущим именем базы, и запрос - добавление элемента в стек

SPUSH(stack, item); // Добавляем элемент в стек

nal\_st += 1; // Увеличиваем счетчик найденных баз на 1

}

if ((nal\_st == 0) && (strcmp(query, "SPUSH") == 0)) { // Если не найдено ни одной базы с заданным именем, и запрос - добавление элемента в стек

line[i][0] = '#'; // Устанавливаем первый символ строки в решетку

for (int j = 1; (basename[j - 1] != 0) && (j < MAX\_SIZE); j++) { // Цикл по символам заданного имени базы

line[i][j] = basename[j - 1]; // Копируем символы заданного имени базы в строку, начиная со второго символа

}

SPUSH(stack, line[i]); // Добавляем строку в стек

SPUSH(stack, item);

}

break;

}

}

fclose(file);

return stack;

}

// Функция для сохранения стека в файл

void SaveStack(Stack\* other, Stack\* stack, const char\* filename, const char\* basename) {

FILE\* file = fopen(filename, "w"); // Открываем файл для записи

if (file == NULL) { // Проверяем, не произошла ли ошибка при открытии файла

printf("Error opening file\n"); // Выводим сообщение об ошибке

return; // Завершаем функцию

}

for (int i = 0; i <= stack->top; i++) { // Цикл по элементам стека

if (i == 0) { // Если это первый элемент стека

fprintf(file, "%s", stack->elements[i]); // Записываем его в файл без переноса строки

}

else { // Если это не первый элемент стека

fprintf(file, "\n%s", stack->elements[i]); // Записываем его в файл с переносом строки

}

}

for (int i = 0; i <= other->top; i++) { // Цикл по элементам другого стека

fprintf(file, "\n%s", other->elements[i]); // Записываем их в файл с переносом строки

}

fclose(file); // Закрываем файл

}

typedef struct {

char\*\* elements; // массив указателей на строки

int front; // индекс первого элемента

int rear; // индекс последнего элемента

} Queue;

Queue\* InitQueue() { // инициализация очереди

Queue\* queue = (Queue\*)malloc(sizeof(Queue)); // выделение памяти под структуру очереди

queue->elements = (char\*\*)malloc(MAX\_SIZE \* sizeof(char\*)); // выделение памяти под массив указателей на строки

queue->front = -1; // начальное значение индекса первого элемента

queue->rear = -1; // начальное значение индекса последнего элемента

return queue; // возврат указателя на структуру очереди

}

void QPUSH(Queue\* queue, char\* element) { // добавление элемента в очередь

if ((queue->rear + 1) % MAX\_SIZE == queue->front) { // проверка на переполнение очереди

printf("Queue full\n"); // вывод сообщения об ошибке

return; // выход из функции

}

if (queue->front == -1) { // если очередь пуста

queue->front = 0; // установка индекса первого элемента

queue->rear = 0; // установка индекса последнего элемента

}

else { // если очередь не пуста

queue->rear = (queue->rear + 1) % MAX\_SIZE; // установка индекса последнего элемента

}

queue->elements[queue->rear] = \_strdup(element); // добавление элемента в очередь

}

void QPOP(Queue\* queue, int output) { // удаление элемента из очереди

if (queue->front == -1) { // проверка на пустоту очереди

printf("Queue is empty\n"); // вывод сообщения об ошибке

return; // выход из функции

}

char\* element = queue->elements[queue->front]; // сохранение значения первого элемента

if (queue->front == queue->rear) { // если в очереди остался один элемент

queue->front = -1; // установка начальных значений индексов

queue->rear = -1;

}

else { // если в очереди осталось более одного элемента

queue->front = (queue->front + 1) % MAX\_SIZE; // установка индекса первого элемента

}

if (output == 1) { // если нужно вывести значение удаленного элемента

printf("item: %s\n", element); // вывод значения удаленного элемента

}

}

Queue\* ExeQueue(Stack\* other, const char\* filename, const char\* basename, const char\* item, const char\* query) { // выполнение команд из файла

FILE\* file = fopen(filename, "r"); // открытие файла на чтение

if (file == NULL) { // проверка на ошибку открытия файла

printf("Error opening file\n"); // вывод сообщения об ошибке

return NULL; // выход из функции

}

Queue\* queue = InitQueue(); // инициализация очереди

char line[MAX\_SIZE][MAX\_SIZE] = { 0 }; // массив строк для хранения команд

char c; // текущий символ

char tempbase[MAX\_SIZE] = { 0 }; // временный буфер для хранения имени базы данных

int sch\_st = 0; int bel = 2; // счетчики

int nal\_st = 0; // счетчик добавленных элементов в очередь

for (int i = 0; i + sch\_st < MAX\_SIZE - 3; i++) { // чтение команд из файла

fscanf(file, "%s", line[i]); // чтение строки из файла

c = line[i][0]; // получение первого символа строки

if (c == '%') { // если строка начинается с символа процента

bel = 1; // установка флаг

bel = 1;

if ((basename != NULL) && (strcmp(tempbase, basename) == 0) && (strcmp(query, "QPUSH") == 0)) {

QPUSH(queue, item);

nal\_st += 1;

sch\_st += 1;

}

for (int v = 0; v < MAX\_SIZE; v++) {

tempbase[v] = 0;

}

for (int j = 1; line[i][j] != '\0'; j++) {

tempbase[j - 1] = line[i][j];

}

}

if (c == '#' || c == '$' || c == '^') {

bel = 0;

}

if (c == '\n') {

continue;

}

if (bel == 1) {

QPUSH(queue, line[i]);

}

if (bel == 0) {

SPUSH(other, line[i]);

}

if (feof(file)) {

if ((basename != NULL) && (strcmp(tempbase, basename) == 0) && (strcmp(query, "QPUSH") == 0)) {

QPUSH(queue, item);

nal\_st += 1;

}

if ((nal\_st == 0) && (strcmp(query, "QPUSH") == 0)) {

line[i][0] = '%';

for (int j = 1; (basename[j - 1] != 0) && (j < MAX\_SIZE); j++) {

line[i][j] = basename[j - 1];

}

QPUSH(queue, line[i]);

QPUSH(queue, item);

}

break;

}

}

fclose(file);

return queue;

}

void SaveQueue(Stack\* other, Queue\* queue, const char\* filename, const char\* basename) {

FILE\* file = fopen(filename, "w");

if (file == NULL) {

printf("Error opening file\n");

return;

}

int n = 0;

while (queue->front != -1) {

char\* element = queue->elements[queue->front];

fprintf(file, "%s\n", element);

n = 1;

QPOP(queue, 0);

}

for (int i = 0; i <= other->top; i++) {

if (n == 1 && i == 0) {

fprintf(file, "%s", other->elements[i]);

}

else {

fprintf(file, "\n%s", other->elements[i]);

}

}

fclose(file);

}

// Структура для представления узла двусвязного списка

typedef struct Node {

char\* element; // Указатель на элемент узла

char\* set\_name; // Указатель на имя множества, к которому принадлежит узел

int hash; // Хеш-значение элемента узла

struct Node\* next; // Указатель на следующий узел в списке

struct Node\* prev; // Указатель на предыдущий узел в списке

} Node;

// Структура для представления множества

typedef struct Set {

Node\* head; // Указатель на голову списка

int size; // Размер множества

Node\* hashTable[MAX\_SIZE]; // Массив указателей на узлы, используемый для хранения элементов множества по хеш-значениям

int tableSize; // Размер массива указателей

int\* emptySlots; // Массив целых чисел, используемый для хранения информации о свободных слотах в массиве указателей

} Set;

// Функция для инициализации множества

Set\* InitSet() {

Set\* set = (Set\*)malloc(sizeof(Set)); // Выделяем память под структуру множества

set->head = NULL; // Устанавливаем указатель на голову списка в NULL

set->size = 0; // Устанавливаем размер множества в 0

set->tableSize = MAX\_SIZE; // Устанавливаем размер массива указателей в MAX\_SIZE

set->emptySlots = (int\*)malloc(MAX\_SIZE \* sizeof(int)); // Выделяем память под массив целых чисел

for (int i = 0; i < MAX\_SIZE; i++) { // Цикл по элементам массива указателей и массива целых чисел

set->hashTable[i] = NULL; // Устанавливаем указатель на узел в NULL

set->emptySlots[i] = 1; // Устанавливаем значение целого числа в 1, что означает свободный слот

}

return set; // Возвращаем указатель на множество

}

// Функция для вычисления хеш-значения элемента множества

int HashSet(char\* element) {

int hash = 0; // Объявляем переменную для хранения хеш-значения

for (int i = 0; element[i] != '\0'; i++) { // Цикл по символам элемента

hash += element[i]; // Прибавляем к хеш-значению код символа

}

return hash % MAX\_SIZE; // Возвращаем остаток от деления хеш-значения на MAX\_SIZE

}

// Функция для добавления элемента в множество

void SADD(Set\* set, char\* element, char\* basename, int need\_key) {

if (strcmp(element, "") == 0) { return; } // Если элемент пустой, то выходим из функции

int unic\_key = 1; // Объявляем переменную для хранения флага уникальности ключа

int hash = HashSet(element) % set->tableSize; // Вычисляем хеш-значение элемента

int temp\_hash = hash; // Сохраняем хеш-значение во временную переменную

while (set->emptySlots[hash] == 0) { // Пока слот в массиве указателей не свободен

if ((HashSet(set->hashTable[hash]->element) == temp\_hash) && (strcmp(set->hashTable[hash]->set\_name, basename)==0)) { // Если хеш-значение элемента в узле совпадает с временным хеш-значением и имя множества в узле совпадает с заданным именем множества

unic\_key = 0; // Устанавливаем флаг уникальности ключа в 0

}

hash = (hash + 1) % set->tableSize; // Переходим к следующему слоту в массиве указателей

}

if (unic\_key == 1) { // Если флаг уникальности ключа равен 1

Node\* newNode = (Node\*)malloc(sizeof(Node)); // Выделяем память под новый узел

newNode->element = \_strdup(element); // Копируем элемент в узел

newNode->set\_name = \_strdup(basename); // Копируем имя множества в узел

newNode->hash = hash; // Сохраняем хеш-значение в узел

newNode->next = set->head; // Устанавливаем указатель на следующий узел в списке в голову списка

if (set->head != NULL) { // Если список не пустой

set->head->prev = newNode; // Устанавливаем указатель на предыдущий узел в голове списка в новый узел

}

set->head = newNode; // Устанавливаем голову списка в новый узел

set->hashTable[hash] = newNode; // Сохраняем указатель на новый узел в массиве указателей по хеш-значению

set->emptySlots[hash] = 0; // Устанавливаем значение целого числа в массиве целых чисел по хеш-значению в 0, что означает занятый слот

set->size++; // Увеличиваем размер множества на 1

}

if (need\_key == 1) { // Если нужно проверить уникальность ключа

need\_unic\_key = unic\_key; // Сохраняем флаг уникальности ключа в глобальную переменную

}

}

// Функция для удаления элемента из множества

void SREM(Set\* set, char\* element, char\* basename) {

int hash = HashSet(element) % set->tableSize; // Вычисляем хеш-значение элемента

while (set->hashTable[hash] != NULL) { // Пока слот в массиве указателей не пустой

if (strcmp(set->hashTable[hash]->element, element) == 0 && strcmp(set->hashTable[hash]->set\_name, basename) == 0) { // Если элемент и имя множества в узле совпадают с заданными

Node\* nodeToRemove = set->hashTable[hash]; // Сохраняем указатель на узел для удаления

if (nodeToRemove == set->head) { // Если узел для удаления - голова списка

set->head = nodeToRemove->next; // Устанавливаем голову списка

}

else {

if (nodeToRemove->prev != NULL) {

nodeToRemove->prev->next = nodeToRemove->next;

}

}

if (nodeToRemove->next != NULL) {

nodeToRemove->next->prev = nodeToRemove->prev;

}

free(nodeToRemove->element);

free(nodeToRemove);

set->hashTable[hash] = NULL;

set->emptySlots[hash] = 1;

set->size--;

}

else {

break;

}

hash = (hash + 1) % set->tableSize;

}

}

.......................................

// Функция для сохранения стека в файл

void SaveStack(Stack\* other, Stack\* stack, const char\* filename, const char\* basename) {

FILE\* file = fopen(filename, "w"); // Открываем файл для записи

if (file == NULL) { // Проверяем, не произошла ли ошибка при открытии файла

printf("Error opening file\n"); // Выводим сообщение об ошибке

return; // Завершаем функцию

}

for (int i = 0; i <= stack->top; i++) { // Цикл по элементам стека

if (i == 0) { // Если это первый элемент стека

fprintf(file, "%s", stack->elements[i]); // Записываем его в файл без переноса строки

}

else { // Если это не первый элемент стека

fprintf(file, "\n%s", stack->elements[i]); // Записываем его в файл с переносом строки

}

}

for (int i = 0; i <= other->top; i++) { // Цикл по элементам другого стека

fprintf(file, "\n%s", other->elements[i]); // Записываем их в файл с переносом строки

}

fclose(file); // Закрываем файл

}

// Структура для представления узла двусвязного списка

typedef struct Node {

char\* element; // Указатель на элемент узла

char\* set\_name; // Указатель на имя множества, к которому принадлежит узел

int hash; // Хеш-значение элемента узла

struct Node\* next; // Указатель на следующий узел в списке

struct Node\* prev; // Указатель на предыдущий узел в списке

} Node;

// Структура для представления множества

typedef struct Set {

Node\* head; // Указатель на голову списка

int size; // Размер множества

Node\* hashTable[MAX\_SIZE]; // Массив указателей на узлы, используемый для хранения элементов множества по хеш-значениям

int tableSize; // Размер массива указателей

int\* emptySlots; // Массив целых чисел, используемый для хранения информации о свободных слотах в массиве указателей

} Set;

// Функция для инициализации множества

Set\* InitSet() {

Set\* set = (Set\*)malloc(sizeof(Set)); // Выделяем память под структуру множества

set->head = NULL; // Устанавливаем указатель на голову списка в NULL

set->size = 0; // Устанавливаем размер множества в 0

set->tableSize = MAX\_SIZE; // Устанавливаем размер массива указателей в MAX\_SIZE

set->emptySlots = (int\*)malloc(MAX\_SIZE \* sizeof(int)); // Выделяем память под массив целых чисел

for (int i = 0; i < MAX\_SIZE; i++) { // Цикл по элементам массива указателей и массива целых чисел

set->hashTable[i] = NULL; // Устанавливаем указатель на узел в NULL

set->emptySlots[i] = 1; // Устанавливаем значение целого числа в 1, что означает свободный слот

}

return set; // Возвращаем указатель на множество

}

// Функция для вычисления хеш-значения элемента множества

int HashSet(char\* element) {

int hash = 0; // Объявляем переменную для хранения хеш-значения

for (int i = 0; element[i] != '\0'; i++) { // Цикл по символам элемента

hash += element[i]; // Прибавляем к хеш-значению код символа

}

return hash % MAX\_SIZE; // Возвращаем остаток от деления хеш-значения на MAX\_SIZE

}

// Функция для добавления элемента в множество

void SADD(Set\* set, char\* element, char\* basename, int need\_key) {

if (strcmp(element, "") == 0) { return; } // Если элемент пустой, то выходим из функции

int unic\_key = 1; // Объявляем переменную для хранения флага уникальности ключа

int hash = HashSet(element) % set->tableSize; // Вычисляем хеш-значение элемента

int temp\_hash = hash; // Сохраняем хеш-значение во временную переменную

while (set->emptySlots[hash] == 0) { // Пока слот в массиве указателей не свободен

if ((HashSet(set->hashTable[hash]->element) == temp\_hash) && (strcmp(set->hashTable[hash]->set\_name, basename)==0)) { // Если хеш-значение элемента в узле совпадает с временным хеш-значением и имя множества в узле совпадает с заданным именем множества

unic\_key = 0; // Устанавливаем флаг уникальности ключа в 0

}

hash = (hash + 1) % set->tableSize; // Переходим к следующему слоту в массиве указателей

}

if (unic\_key == 1) { // Если флаг уникальности ключа равен 1

Node\* newNode = (Node\*)malloc(sizeof(Node)); // Выделяем память под новый узел

newNode->element = \_strdup(element); // Копируем элемент в узел

newNode->set\_name = \_strdup(basename); // Копируем имя множества в узел

newNode->hash = hash; // Сохраняем хеш-значение в узел

newNode->next = set->head; // Устанавливаем указатель на следующий узел в списке в голову списка

if (set->head != NULL) { // Если список не пустой

set->head->prev = newNode; // Устанавливаем указатель на предыдущий узел в голове списка в новый узел

}

set->head = newNode; // Устанавливаем голову списка в новый узел

set->hashTable[hash] = newNode; // Сохраняем указатель на новый узел в массиве указателей по хеш-значению

set->emptySlots[hash] = 0; // Устанавливаем значение целого числа в массиве целых чисел по хеш-значению в 0, что означает занятый слот

set->size++; // Увеличиваем размер множества на 1

}

if (need\_key == 1) { // Если нужно проверить уникальность ключа

need\_unic\_key = unic\_key; // Сохраняем флаг уникальности ключа в глобальную переменную

}

}

// Функция для удаления элемента из множества

void SREM(Set\* set, char\* element, char\* basename) {

// Вычисляем хэш-код элемента

int hash = HashSet(element) % set->tableSize;

// Пока не найдем элемент или пустую ячейку

while (set->hashTable[hash] != NULL) {

// Если нашли элемент

if (strcmp(set->hashTable[hash]->element, element) == 0 && strcmp(set->hashTable[hash]->set\_name, basename) == 0) {

// Удаляем элемент из списка

Node\* nodeToRemove = set->hashTable[hash];

if (nodeToRemove == set->head) {

set->head = nodeToRemove->next;

}

else {

if (nodeToRemove->prev != NULL) {

nodeToRemove->prev->next = nodeToRemove->next;

}

}

if (nodeToRemove->next != NULL) {

nodeToRemove->next->prev = nodeToRemove->prev;

}

// Освобождаем память

free(nodeToRemove->element);

free(nodeToRemove);

// Удаляем элемент из хэш-таблицы

set->hashTable[hash] = NULL;

set->emptySlots[hash] = 1;

set->size--;

}

else {

break;

}

// Переходим к следующей ячейке

hash = (hash + 1) % set->tableSize;

}

}

int SISMEMBER(Set\* set, char\* element, char\* basename) {

// Вычисляем хэш-код элемента

int hash = HashSet(element) % set->tableSize;

// Ищем элемент в хэш-таблице

for (int i = 0; i < MAX\_SIZE; i++) {

Node\* current = set->hashTable[hash];

if ((current != NULL) && ((current->element != 0) && (current->set\_name != 0)) && (strcmp(current->element, element) == 0 && strcmp(current->set\_name, basename) == 0)) {

return 1;

}

if (current == NULL) {

return 0;

}

// Переходим к следующей ячейке

hash = (hash + 1) % set->tableSize;

}

return 0;

}

Set\* ExeSet(Stack\* other, char\* filename, char\* basename, char\* item, char\* query) {

// Открываем файл

FILE\* file = fopen(filename, "r");

if (file == NULL) {

printf("Error opening file\n");

return NULL;

}

// Инициализируем множество

Set\* set = InitSet();

// Считываем строки из файла

char line[MAX\_SIZE][MAX\_SIZE] = { 0 };

int sch\_set = 0; int nal\_set = 0; int sch\_n = 0; int sch = 0; int bel = 2;

char c; char tempbase[MAX\_SIZE] = { 0 };

for (int i = 0; i + sch\_set < MAX\_SIZE - 3; i++) {

fscanf(file, "%s", line[i]);

c = line[i][0];

// Если строка начинается с символа '$'

if (c == '$') {

bel = 1;

// Если это команда SADD и база данных совпадает с заданной

if ((strcmp(tempbase, basename) == 0) && (strcmp(query, "SADD") == 0)) {

// Добавляем элемент в множество

SADD(set, item, basename, 1);

nal\_set += 1;

sch\_set += 1;

}

for (int v = 0; v < MAX\_SIZE; v++) {

tempbase[v] = 0;

}

for (int j = 1; line[i][j] != '\0'; j++) {

tempbase[j - 1] = line[i][j];

}

}

if (c == '#' || c == '%' || c == '^') {

bel = 0;

}

if (c == '\n') {

sch\_n += 1;

continue;

}

if (bel == 1) {

if ((c != '$') && (c != '\n')) {

SADD(set, line[i], tempbase, 0);

}

}

if (bel == 0) {

SPUSH(other, line[i]);

}

if (feof(file)) {

if ((strcmp(tempbase, basename) == 0) && (strcmp(query, "SADD") == 0)) {

SADD(set, item, basename, 1);

nal\_set += 1;

}

if ((nal\_set == 0) && (strcmp(query, "SADD") == 0)) {

SADD(set, item, basename, 1);

}

break;

}

}

fclose(file);

return set;

}

void SaveSet(Stack\* other, Set\* set, const char\* filename) {

FILE\* file = fopen(filename, "w"); // Открываем файл для записи

if (file == NULL) { // Проверяем, удалось ли открыть файл

printf("Ошибка открытия файла\n"); // Если не удалось, выводим сообщение об ошибке

return; // И завершаем функцию

}

int n = 0; // Флаг, который показывает, были ли записаны элементы множества в файл

while (1) { // Бесконечный цикл

Node\* current = set->head; // Указатель на текущий элемент множества

if (current == NULL || current->element == NULL || current->set\_name == NULL) { goto jump; } // Если текущий элемент равен NULL или его поля element и set\_name равны NULL, переходим к метке jump

Node\* pred = current; // Указатель на предыдущий элемент множества

char\* basename = current->set\_name; // Имя множества

fprintf(file, "$%s\n", basename); // Записываем имя множества в файл

while (current != NULL) { // Цикл по всем элементам множества

if (strcmp(basename, current->set\_name) == 0) { // Если имя текущего элемента совпадает с именем множества

fprintf(file, "%s\n", current->element); // Записываем элемент в файл

n = 1; // Устанавливаем флаг, что были записаны элементы множества в файл

Node\* vrem = current->next; // Указатель на следующий элемент множества

if (set->head == current) { set->head = vrem; } // Если текущий элемент является головой множества, устанавливаем в качестве головы следующий элемент

current->element = NULL; // Удаляем элемент из множества

current->set\_name = NULL; // Удаляем имя множества

pred->next = current->next; // Удаляем текущий элемент из списка

current = NULL; // Устанавливаем текущий элемент в NULL

current = vrem; // Устанавливаем текущим элементом следующий элемент

}

else {

pred = current; // Переходим к следующему элементу

current = current->next;

}

}

}

jump:

for (int i = 0; i <= other->top; i++) { // Цикл по всем элементам стека other

if (n == 1 && i == 0) { // Если были записаны элементы множества в файл и текущий элемент является первым элементом стека

fprintf(file, "%s", other->elements[i]); // Записываем элемент в файл без перевода строки

}

else {

fprintf(file, "\n%s", other->elements[i]); // Записываем элемент в файл с переводом строки

}

}

fclose(file); // Закрываем файл

}

typedef struct entry {

char\* key; // Ключ

char\* value; // Значение

struct entry\* next; // Указатель на следующий элемент

struct entry\* prev; // Указатель на предыдущий элемент

struct entry\* coll\_next; // Указатель на следующий элемент в коллизионной цепочке

struct entry\* coll\_prev; // Указатель на предыдущий элемент в коллизионной цепочке

char\* tableName; // Имя таблицы

} Entry;

typedef struct {

Entry\* elements; // Массив элементов

Entry\* hashTable[MAX\_SIZE]; // Хэш-таблица

int size; // Размер таблицы

} HashTable;

int HashforTable(char\* key, int size) { // Функция хэширования

int sum = 0; // Сумма

for (int i = 0; key[i] != '\0'; i++)

sum += key[i];

}

return sum % size;

}

HashTable\* InitTable(int size) {

HashTable\* table = (HashTable\*)malloc(sizeof(HashTable)); // Выделяем память под хэш-таблицу

table->elements = (Entry\*)malloc(size \* sizeof(Entry)); // Выделяем память под элементы таблицы

for (int i = 0; i < MAX\_SIZE; i++) {

table->hashTable[i] = NULL; // Инициализируем хэш-таблицу

}

table->elements = NULL; // Устанавливаем элементы таблицы в NULL

table->size = size; // Устанавливаем размер таблицы

return table; // Возвращаем указатель на хэш-таблицу

}

void HSET(HashTable\* table, char\* key, char\* value, char\* tableName, int need\_key) {

int index = HashforTable(key, table->size); // Вычисляем индекс элемента в таблице

Entry\* entry = (Entry\*)malloc(sizeof(Entry)); // Выделяем память под элемент

entry->key = (char\*)malloc(strlen(key) + 1); // Выделяем память под ключ

strcpy(entry->key, key); // Копируем ключ

entry->value = (char\*)malloc(strlen(value) + 1); // Выделяем память под значение

strcpy(entry->value, value); // Копируем значение

entry->tableName = (char\*)malloc(strlen(tableName) + 1); // Выделяем память под имя таблицы

strcpy(entry->tableName, tableName); // Копируем имя таблицы

entry->next = NULL; // Устанавливаем указатель на следующий элемент в NULL

entry->prev = NULL; // Устанавливаем указатель на предыдущий элемент в NULL

entry->coll\_next = NULL; // Устанавливаем указатель на следующий элемент в коллизионной цепочке в NULL

entry->coll\_prev = NULL; // Устанавливаем указатель на предыдущий элемент в коллизионной цепочке в NULL

Entry\* current = table->elements; // Указатель на текущий элемент таблицы

int unic\_key = 0; // Флаг, который показывает, уникален ли ключ

Entry\* temp = table->hashTable[index]; // Указатель на элемент в хэш-таблице

if (table->hashTable[index] == NULL) {

unic\_key = 1; // Если элемент в хэш-таблице равен NULL, ключ уникален

}

while ((temp != NULL) && (strcmp(temp->key, entry->key) != 0)) { // Цикл по всем элементам в коллизионной цепочке

temp = temp->coll\_prev; // Переходим к предыдущему элементу в коллизионной цепочке

if (temp == NULL) {

unic\_key = 1; // Если элемент в коллизионной цепочке равен NULL, ключ уникален

}

}

if (unic\_key == 1) { // Если ключ уникален

if (current == NULL) {

table->elements = entry; // Если текущий элемент равен NULL, устанавливаем в качестве текущего элемента entry

}

else {

entry->next = current; // Устанавливаем указатель на следующий элемент entry на текущий элемент

current->prev = entry; // Устанавливаем указатель на предыдущий элемент текущего элемента на entry

table->elements = entry; // Устанавливаем в качестве текущего элемента entry

}

if (table->hashTable[index] == NULL) {

table->hashTable[index] = entry; // Если элемент в хэш-таблице равен NULL, устанавливаем в качестве элемента entry

}

else {

entry->coll\_next = table->hashTable[index]; // Устанавливаем указатель на следующий элемент entry на элемент в хэш-таблице

table->hashTable[index]->coll\_prev = entry; // Устанавливаем указатель на предыдущий

table->hashTable[index] = entry;

}

}

if (need\_key == 1) {

need\_unic\_key = unic\_key;

}

}

int HFIND(HashTable\* table, char\* basename, char\* item, char\* key) {

int index = HashforTable(key, table->size); // Вычисление индекса для данного ключа

Entry\* current = table->hashTable[index]; // Получение указателя на первую запись в списке с этим индексом

while (1) { // Бесконечный цикл

if (current == NULL) { // Если узел пустой, значит элемент не найден

return 0; // Возвращаем 0

}

if ((current != NULL) && // Если текущий узел не является пустым

((current->key != NULL) && (current->tableName != NULL) && (current->value != NULL)) && // И его поля key, tableName и value не равны NULL

(strcmp(current->key, key) == 0) && // И ключ текущего узла совпадает с искомым ключом

(strcmp(current->tableName, basename) == 0) && // И tableName текущего узла совпадает с искомым basename

(strcmp(current->value, item) == 0)) { // И value текущего узла совпадает с искомым item

return 1; // Возвращаем 1, что означает, что элемент найден

}

else { // Если условия не выполнены, переходим к следующему узлу в цепочке коллизий

current = current->coll\_next; // Переходим к следующему узлу

}

}

}

char\* HGET(HashTable\* table, char\* basename, char\* key) {

int index = HashforTable(key, table->size); // Вычисление индекса для данного ключа

Entry\* current = table->hashTable[index]; // Получение указателя на первую запись в списке с этим индексом

while (1) { // Бесконечный цикл

if (current == NULL) { // Если узел пустой, значит элемент не найден

return NULL; // Возвращаем NULL

}

if ((current != NULL) && // Если текущий узел не является пустым

((current->key != NULL) && (current->tableName != NULL) && (current->value != NULL)) && // И его поля key, tableName и value не равны NULL

(strcmp(current->key, key) == 0) && // И ключ текущего узла совпадает с искомым ключом

(strcmp(current->tableName, basename) == 0)) { // И tableName текущего узла совпадает с искомым basename

return current->value; // Возвращаем значение value текущего узла

}

else { // Если условия не выполнены, переходим к следующему узлу в цепочке коллизий

current = current->coll\_next; // Переходим к следующему узлу

}

}

}

// Функция для удаления элемента из хеш-таблицы

int HDEL(HashTable\* table, char\* basename, char\* item, char\* key) {

// Проверяем, есть ли такой элемент в таблице

if (HFIND(table, basename, item, key) == 1) {

// Вычисляем индекс хеша для ключа

int index = HashforTable(key, table->size);

// Получаем текущий элемент с данным индексом

Entry\* current = table->hashTable[index];

Entry\* vrem = NULL;

// Проходим по всем элементам с данного индекса

while (current != NULL) {

// Проверяем, совпадают ли значения ключа, имени таблицы и значения с текущим элементом

if ((current->key != NULL) && (current->tableName != NULL) && (current->value != NULL) && (strcmp(current->key, key) == 0) && (strcmp(current->tableName, basename) == 0) && (strcmp(current->value, item) == 0)) {

// Если текущий элемент является первым элементом в списке

if (table->elements == current) {

table->elements = current->next;

}

// Иначе

else {

// Присваиваем предыдущему элементу указатель на следующий элемент

current->prev->next = current->next;

// Если следующий элемент существует, присваиваем ему указатель на предыдущий элемент

if (current->next != NULL) { current->next->prev = current->prev; }

}

// Если нет элементов с тем же индексом

if (current->coll\_prev == NULL) {

// Присваиваем хеш-таблице указатель на следующий элемент

table->hashTable[index] = current->coll\_next;

// Присваиваем временной переменной текущий коллизионный элемент

vrem = current->coll\_next;

// Очищаем память, занятую текущим элементом

current->key = NULL;

current->value = NULL;

current->tableName = NULL;

free(current);

}

// Иначе

else {

// Присваиваем предыдущему коллизионному элементу указатель на следующий коллизионный элемент

current->coll\_prev->coll\_next = current->coll\_next;

// Присваиваем временной переменной текущий коллизионный элемент

vrem = current->coll\_next;

// Очищаем память, занятую текущим элементом

current->key = NULL;

current->value = NULL;

current->tableName = NULL;

free(current);

}

// Переходим к следующему элементу

current = vrem;

}

else {

// Присваиваем текущему элементу указатель на себя

current->coll\_prev = current;

// Переходим к следующему элементу

current = current->coll\_next;

}

}

// Возвращаем 1 для успешного удаления

return 1;

}

// Иначе, возвращаем 0

else {

return 0;

}

}

// Функция, которая выполняет обработку файла и заполняет хэш-таблицу

// на основе данных из файла

HashTable\* ExeTable(Stack\* other, char\* filename, char\* basename, char\* item, char\* key, char\* query) {

FILE\* file = fopen(filename, "r"); // Открытие файла в режиме чтения

HashTable\* table = InitTable(MAX\_SIZE); // Инициализация хэш-таблицы

char line[MAX\_SIZE][MAX\_SIZE] = { 0 }; // Массив для хранения строк из файла

char c;

char tempbase[MAX\_SIZE] = { 0 }; // Временная переменная для хранения имени базы данных

int bel = 2; // Флаг, указывающий, в какой части файла находимся

int sch\_hash = 0; // Счетчик строк, соответствующих базе данных по имени basename

int nal\_hash = 0; // Счетчик строк, соответствующих определенному ключу key

char temp\_key[MAX\_SIZE] = { 0 }; // Временная переменная для хранения ключа

char temp\_line[MAX\_SIZE] = { 0 }; // Временная переменная для хранения значения

int flag = 0; // Флаг, указывающий, была ли уже обработана строка с ключом key и базой данных basename

// Чтение строк из файла и их обработка

for (int i = 0; i + sch\_hash < MAX\_SIZE - 3; i++) {

fscanf(file, "%s", line[i]); // Чтение строки из файла

c = line[i][0]; // Получение первого символа строки

if (c == '^') { // Если первый символ - '^', то находимся в строке с именем базы данных

bel = 1; // Установка флага bel в значение 1

// Проверка, является ли текущая строка строкой с именем базы данных и запросом "HSET"

if ((strcmp(tempbase, basename) == 0) && (strcmp(query, "HSET") == 0)) {

HSET(table, key, item, basename, 1); // Вызов функции HSET и добавление элемента в хэш-таблицу

flag = 1; // Установка флага flag в значение 1

nal\_hash += 1; // Увеличение счетчика nal\_hash на 1

sch\_hash += 1; // Увеличение счетчика sch\_hash на 1

}

for (int a = 1; a < MAX\_SIZE; a++) {

tempbase[a] = 0; // Очистка временной переменной tempbase

}

// Копирование символов из строки line[i] в tempbase, начиная со второго символа

for (int j = 1; line[i][j] != '\0'; j++) {

tempbase[j - 1] = line[i][j];

}

// Очистка временных переменных temp\_key и temp\_line

for (int v = 0; v < MAX\_SIZE; v++) {

temp\_key[v] = 0;

}

for (int f = 0; f < MAX\_SIZE; f++) {

temp\_line[f] = 0;

}

}

// Если первый символ строки - '#' или '$' или '%', то находимся в обычной строке

if (c == '#' || c == '$' || c == '%') {

bel = 0; // Установка флага bel в значение 0

}

// Если первый символ строки - символ новой строки, то пропускаем строку

if (c == '\n') {

continue;

}

// Если bel равно 1, то находимся в строке с именем базы данных

if (bel == 1) {

// Если первый символ строки - '~', то находимся в строке с ключом и значением

if (c == '~') {

// Очистка временной переменной temp\_key

for (int v = 0; v < MAX\_SIZE; v++) {

temp\_key[v] = 0;

}

int j = 1;

// Копирование символов из строки line[i] в temp\_key, начиная со второго символа

for (; line[i][j] != '~' && j < MAX\_SIZE; j++) {

temp\_key[j - 1] = line[i][j];

}

// Очистка временной переменной temp\_line

for (int f = 0; f < MAX\_SIZE; f++) {

temp\_line[f] = 0;

}

// Копирование символов из строки line[i] в temp\_line, начиная с j+1 символа

for (int g = j + 1; g < MAX\_SIZE; g++) {

temp\_line[g - j - 1] = line[i][g];

}

}

// Если первый символ строки - '~', то вызываем функцию HSET и добавляем элемент в хэш-таблицу

if (c =='~') {

HSET(table, temp\_key, temp\_line, tempbase, 0);

}

}

if (bel == 0) {

SPUSH(other, line[i]);

}

if (feof(file)) {

if ((strcmp(tempbase, basename) == 0) && (strcmp(query, "HSET") == 0) && (flag == 0)) {

HSET(table, key, item, basename, 1);

nal\_hash += 1;

}

if ((nal\_hash == 0) && (strcmp(query, "HSET") == 0)) {

HSET(table, key, item, basename, 1);

}

break;

}

}

fclose(file);

return table;

}

void SaveTable(Stack\* other, HashTable\* table, char\* filename) {

// Открыть файл для записи

FILE\* file = fopen(filename, "w");

// Проверить, удалось ли открыть файл

if (file == NULL) {

printf("Ошибка открытия файла\n");

return;

}

int n = 0;

// Пока в таблице есть элементы

while (table->elements != NULL) {

// Получить текущий элемент таблицы

Entry\* current = table->elements;

// Если текущий элемент имеет пустые поля, выйти из цикла

if (current == NULL || current->value == NULL || current->tableName == NULL || current->key == NULL) {

break;

}

// Сохранить указатель на текущий элемент

Entry\* pred = current;

// Сохранить указатель на имя таблицы текущего элемента

char\* basename = current->tableName;

// Записать имя таблицы в файл

fprintf(file, "^%s\n", basename);

// Пока не достигнут конец цепочки элементов с одинаковым именем таблицы

while (current != NULL) {

// Если имя таблицы текущего элемента не пустое и совпадает с сохраненным именем

if (current->tableName != 0 && strcmp(basename, current->tableName) == 0) {

// Записать ключ и значение текущего элемента в файл

fprintf(file, "~%s~%s\n", current->key, current->value);

// Установить флаг наличия данных в файле

n = 1;

// Сохранить указатель на следующий элемент цепочки

Entry\* vrem = current->next;

// Если текущий элемент - первый в цепочке, обновить указатель на первый элемент

if (table->elements == current) {

table->elements = vrem;

}

// Очистить поля текущего элемента

current->value = NULL;

current->tableName = NULL;

// Удалить текущий элемент из цепочки

pred->next = current->next;

current = NULL;

current = vrem;

}

else {

// Перейти к следующему элементу цепочки

pred = current;

current = current->next;

}

}

}

// Записать данные из стека other в файл

for (int i = 0; i <= other->top; i++) {

if (n == 1 && i == 0) {

fprintf(file, "%s", other->elements[i]);

}

else {

fprintf(file, "\n%s", other->elements[i]);

}

}

// Закрыть файл

fclose(file);

}

// Объявление функции handle\_client, которая будет запускаться параллельно с основным потоком

DWORD WINAPI handle\_client(LPVOID lpParam) {

SOCKET ClientSock = (SOCKET)lpParam; // Приведение типа указателя на сокет клиента, полученного из параметров функции, к типу SOCKET

while (1) { // добавили цикл while, который будет повторяться, пока сокет клиента не будет закрыт, или пока не произойдет ошибка

WaitForSingleObject(mutex, INFINITE);

char\* buffer = (char\*)malloc(BUFFER\_SIZE \* sizeof(char));

int bytes\_received; // добавили переменную для хранения количества байт, полученных от клиента

do { // добавили цикл do-while, который будет повторяться, пока не будет получен весь запрос от клиента, или пока не произойдет ошибка

// получаем часть запроса и сохраняем ее в буфере

bytes\_received = recv(ClientSock, buffer, BUFFER\_SIZE, 0);

if (bytes\_received == SOCKET\_ERROR) { // проверяем, что не произошла ошибка

printf("recv failed: %d\n", WSAGetLastError());

break;

}

if (bytes\_received == 0) { // проверяем, что клиент не отключился

printf("client disconnected\n");

break;

}

} while (bytes\_received == BUFFER\_SIZE); // повторяем, пока не получим меньше, чем размер буфера, что означает, что запрос закончился

if (bytes\_received <= 0) { // если произошла ошибка или отключение, выходим из цикла

break;

} // повторяем, пока не получим меньше, чем размер буфера, что означает, что запрос закончился

buffer[bytes\_received] = '\0'; // добавляем нулевой символ в конец буфера, чтобы обозначить конец строки

// дальше обрабатываем запрос, как вы делали раньше, используя буфер как источник данных

char\* argv[500] = {0};

int argc = 0;

char delimiters[] = " ";

char\* word = strtok(buffer, delimiters);

while (word != NULL) {

argv[argc++] = word;

word = strtok(NULL, delimiters);

} char\* filename = NULL; // Объявление указателя на имя файла

char\* key = NULL; // Объявление указателя на ключ

char\* query = NULL; // Объявление указателя на запрос

char\* item = NULL; // Объявление указателя на элемент

char\* basename = NULL; // Объявление указателя на базовое имя

int hdel = 0; // Объявление переменной, хранящей информацию об удалении

char\* exodus = NULL; // Объявление указателя на строку, хранящую сообщение о выходе из функции

if (((argc == 4) || (argc == 6) || (argc == 7)) && (strcmp(argv[0], "--file") == 0) && (strcmp(argv[2], "--query") == 0)) { // Проверка условий корректности данных

filename = argv[1]; // Присвоение указателю на имя файла адреса имени файла

if ((argc == 6) && (strcmp(argv[3], "HGET") != 0)) { // Если число аргументов равно 6 и первый аргумент не равен "HGET"

query = argv[3]; // Присвоение указателю на запрос адреса запроса

basename = argv[4]; // Присвоение указателю на базовое имя адреса базового имени

item = argv[5]; // Присвоение указателю на элемент адреса элемента

}

else if (argc == 4) { // Если число аргументов равно 4

query = argv[3]; // Присвоение указателю на запрос адреса запроса

}

else if ((argc == 6) && (strcmp(argv[3], "HGET") == 0)) { // Если число аргументов равно 6 и первый аргумент равен "HGET"

query = argv[3]; // Присвоение указателю на запрос адреса запроса

basename = argv[4]; // Присвоение указателю на базовое имя адреса базового имени

key = argv[5]; // Присвоение указателю на ключ адреса ключа

}

else if (argc == 7) { // Если число аргументов равно 7

query = argv[3]; // Присвоение указателю на запрос адреса запроса

basename = argv[4]; // Присвоение указателю на базовое имя адреса базового имени

item = argv[6]; // Присвоение указателю на элемент адреса элемента

key = argv[5]; // Присвоение указателю на ключ адреса ключа

}

else if ((argc != 4) && (argc != 6) && (argc != 7)) { // Если число аргументов не равно 4, 6 и 7

exodus = malloc(200); // Выделение памяти под строку

sprintf(exodus, "Error in number of arguments in query\n"); // Форматирование строки с сообщением об ошибке

return 0; // Выход из функции

}

int len = strlen(filename);

char\* last\_four = NULL;

if (len > 4) { last\_four = &filename[len - 4]; }

if (strcmp(last\_four, ".txt") != 0) {

exodus = malloc(200);

sprintf(exodus, "File name error (please use a text file)\n"); return 0;

}

FILE\* file;

file = fopen(filename, "r");

if (file == NULL) {

file = fopen(filename, "w");

}

if ((strcmp(query, "SPUSH") == 0) && (argc == 6)) {

Stack\* other = InitStack();

Stack\* stack = ExeStack(other, filename, basename, item, query);

fclose(file);

SaveStack(other, stack, filename, basename);

exodus = malloc(strlen(item) + strlen(basename) + 100);

sprintf(exodus, "basename: %s, item: %s", basename, item);

}

if ((strcmp(query, "SPOP") == 0) && (argc == 4)) {

Stack\* other = InitStack();

Stack\* stack = ExeStack(other, filename, basename, item, query);

char\* vrem = NULL;

vrem = SPOP(stack);

exodus = malloc(strlen(vrem) + 100);

sprintf(exodus, "item or basename:%s\n", vrem);

fclose(file);

SaveStack(other, stack, filename, basename); }

if ((strcmp(query, "QPUSH") == 0) && (argc == 6)) {

// Инициализация стека

Stack\* other = InitStack();

// Выполнение операции очереди и сохранение результата

Queue\* queue = ExeQueue(other, filename, basename, item, query);

fclose(file);

SaveQueue(other, queue, filename, basename);

// Формирование сообщения о выполненной операции

exodus = malloc(strlen(item) + strlen(basename) + 100);

sprintf(exodus, "basename: %s, item: %s", basename, item);

}

if ((strcmp(query, "QPOP") == 0) && (argc == 4)) {

// Инициализация стека

Stack\* other = InitStack();

// Выполнение операции извлечения из очереди и сохранение результата

Queue\* queue = ExeQueue(other, filename, basename, item, query);

// Извлечение элемента из очереди

char\* vrem =QPOP(queue, 1);

fclose(file);

SaveQueue(other, queue, filename, basename);

// Формирование сообщения о выполненной операции

exodus = malloc(strlen(vrem)+ 100);

sprintf(exodus, "basename or item: %s", vrem);

}

if ((strcmp(query, "SADD") == 0) && (argc == 6)) {

// Инициализация стека

Stack\* other = InitStack();

// Выполнение операции добавления элемента в множество и сохранение результата

Set\* set = ExeSet(other, filename, basename, item, query);

// Проверка необходимости уникального ключа

if (need\_unic\_key == 1) {

// Формирование сообщения о выполненной операции

exodus = malloc(strlen(item) + strlen(basename) + 100);

sprintf(exodus, "basename: %s, item: %s", basename, item);

} else {

// Формирование сообщения об ошибке, если ключ уже используется

exodus = malloc(200);

sprintf(exodus, "This key is already in use, please enter another key\n");

}

SaveSet(other, set, filename); // Сохранение состояния множества

fclose(file);

}

if ((strcmp(query, "SREM") == 0) && (argc == 6)) {

// Инициализация стека

Stack\* other = InitStack();

// Выполнение операции удаления элемента из множества и сохранение результата

Set\* set = ExeSet(other, filename, basename, item, query);

// Проверка наличия элемента в множестве перед удалением

if (SISMEMBER(set, item, basename)) {

SREM(set, item, basename); // Удаление элемента из множества

// Формирование сообщения о выполненной операции

exodus = malloc(strlen(item) + strlen(basename) + 100);

sprintf(exodus, "basename:%s, element:%s\n", basename, item);

} else {

// Формирование сообщения об ошибке, если элемент отсутствует

exodus = malloc(200);

sprintf(exodus, "%s\n", "Error, element missing");

}

SaveSet(other, set, filename); // Сохранение состояния множества

fclose(file);

}

if ((strcmp(query, "SISMEMBER") == 0) && (argc == 6)) {

// Инициализация стека

Stack\* other = InitStack();

// Выполнение операции проверки наличия элемента в множестве и сохранение результата

Set\* set = ExeSet(other, filename, basename, item, query);

if (SISMEMBER(set, item, basename)) {

// Формирование сообщения о результате проверки

exodus = malloc(200);

sprintf(exodus, "%s\n", "TRUE");

}

else {

// Формирование сообщения о результате проверки

exodus = malloc(200);

sprintf(exodus, "%s\n", "FALSE");

}

SaveSet(other, set, filename); // Сохранение состояния множества

fclose(file);

}

if ((strcmp(query, "HSET") == 0) && (argc == 7)) {

// Инициализация стека

Stack\* other = InitStack();

// Выполнение операции добавления элемента в хэш-таблицу

HashTable\* table = ExeTable(other, filename, basename, item, key, query);

SaveTable(other, table, filename); // Сохранение состояния хэш-таблицы

fclose(file);

// Проверка необходимости уникального ключа

if (need\_unic\_key == 1) {

// Формирование сообщения о выполненной операции

exodus = malloc(strlen(key)+strlen(item)+strlen(basename)+100);

sprintf(exodus, "key: %s, item: %s, basename: %s\n", key, item, basename);

}

else {

// Формирование сообщения об ошибке, если ключ уже используется

exodus = malloc(200);

sprintf(exodus, "This key is already in use, please enter another key\n");

}

}

// Проверяем, если команда является "HDEL" и количество аргументов равно 7

if ((strcmp(query, "HDEL") == 0) && (argc == 7)) {

// Инициализируем стек и хэш-таблицу

Stack\* other = InitStack();

HashTable\* table = ExeTable(other, filename, basename, item, key, query);

// Проверяем, был ли удален элемент из хэш-таблицы и формируем сообщение

if (HDEL(table, basename, item, key)) {

exodus = malloc(strlen(key) + strlen(item) + strlen(basename) + 100);

sprintf(exodus, "key: %s, item: %s, basename: %s\n", key, item, basename);

}

else {

exodus = malloc(200);

sprintf(exodus, "%s", "Error, element was not deleted\n");

}

// Сохраняем измененную таблицу в файл и закрываем файл

SaveTable(other, table, filename);

fclose(file);

}

// Проверяем, если команда является "HGET" и количество аргументов равно 6

if ((strcmp(query, "HGET") == 0) && (argc == 6)) {

// Инициализируем стек и хэш-таблицу

Stack\* other = InitStack();

HashTable\* table = ExeTable(other, filename, basename, item, key, query);

char\* value = NULL;

value = HGET(table, basename, key);

// Проверяем, найдено ли значение в хэш-таблице и формируем сообщение

if (value == NULL) {

exodus = malloc(200);

sprintf(exodus, "False\n");

}

else {

exodus = malloc(strlen(value) + 10);

sprintf(exodus, "%s\n", value);

}

// Сохраняем измененную таблицу в файл и закрываем файл

SaveTable(other, table, filename);

fclose(file);

}

// Если команда не соответствует требуемым условиям, формируем сообщение об ошибке

else {

exodus = malloc(200);

sprintf(exodus, "Invalid command syntax\n");

}

// Отправляем ответ на клиентский сокет

int bytes\_sent = send(ClientSock, exodus, strlen(exodus), 0);

// Освобождаем выделенную память

free(exodus);

free(buffer);

// Закрываем клиентский сокет и освобождаем мьютекс

ReleaseMutex(mutex);

}

closesocket(ClientSock);

}

// Основная функция программы

int main() {

WSADATA wsaData;

SOCKET ServerSock;

SOCKADDR\_IN ServerInfo;

int exodus;

// Инициализируем библиотеку Winsock

exodus = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (exodus != 0) {

printf("WSAStartup failed: %d\n", exodus);

return 1;

}

// Создаем сокет для сервера

ServerSock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (ServerSock == INVALID\_SOCKET) {

printf("socket failed: %d\n", WSAGetLastError());

WSACleanup();

return 1;

}

// Привязываем сокет к адресу и порту

ServerInfo.sin\_family = AF\_INET;

ServerInfo.sin\_addr.s\_addr = htonl(INADDR\_ANY);

ServerInfo.sin\_port = htons(PORT);

exodus = bind(ServerSock, (SOCKADDR\*)&ServerInfo, sizeof(ServerInfo));

if (exodus == SOCKET\_ERROR) {

printf("bind failed: %d\n", WSAGetLastError());

closesocket(ServerSock);

WSACleanup();

return 1;

}

// Переводим сокет в режим прослушивания

exodus = listen(ServerSock, BACKLOG);

if (exodus == SOCKET\_ERROR) {

printf("listen failed: %d\n", WSAGetLastError());

closesocket(ServerSock);

WSACleanup();

return 1;

}

mutex = CreateMutex(NULL, FALSE, NULL);

// Принимаем входящие соединения от клиентов

printf("Server is listening on port %d\n", PORT);

while (1) {

SOCKET ClientSock;

SOCKADDR\_IN ClientAddr;

int ClientAddrSize = sizeof(ClientAddr);

// Принимаем соединение от клиента

ClientSock = accept(ServerSock, (SOCKADDR\*)&ClientAddr, &ClientAddrSize);

if (ClientSock == INVALID\_SOCKET) {

printf("accept failed: %d\n", WSAGetLastError());

closesocket(ServerSock);

WSACleanup();

return 1;

}

// Создаем новый поток для обработки запроса клиента

printf("Accepted connection from %s:%d\n", inet\_ntoa(ClientAddr.sin\_addr), ntohs(ClientAddr.sin\_port));

DWORD ThreadID;

HANDLE ThreadHandle = CreateThread(NULL, 0, handle\_client, (LPVOID)ClientSock, 0, &ThreadID);

if (ThreadHandle == NULL) {

printf("CreateThread failed: %d\n", GetLastError());

closesocket(ClientSock);

closesocket(ServerSock);

WSACleanup();

return 1;

}

CloseHandle(ThreadHandle);

}

CloseHandle(mutex);

// Закрываем сокет сервера и очищаем библиотеку Winsock

closesocket(ServerSock);

WSACleanup();

return 0;

}

**Клиент:**

#define \_CRT\_SECURE\_NO\_WARNINGS

#define \_WINSOCK\_DEPRECATED\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 1000

#include <winsock2.h>

#include <ws2tcpip.h>

#include <windows.h>

#pragma comment(lib, "ws2\_32.lib")

#define PORT 6379

#define BUFFER\_SIZE 104857600

int main() {

WSADATA wsaData;

SOCKET ClientSock;

SOCKADDR\_IN ServerInfo;

int exodus;

// Инициализируем библиотеку Winsock

exodus = WSAStartup(MAKEWORD(2, 2), &wsaData);

if (exodus != 0) {

printf("WSAStartup failed: %d\n", exodus);

return 1;

}

// Создаем сокет для клиента

ClientSock = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP);

if (ClientSock == INVALID\_SOCKET) {

printf("socket failed: %d\n", WSAGetLastError());

WSACleanup();

return 1;

}

// Подключаемся к серверу

ZeroMemory(&ServerInfo, sizeof(ServerInfo));

ServerInfo.sin\_family = AF\_INET;

ServerInfo.sin\_addr.s\_addr = inet\_addr("127.0.0.1"); // адрес сервера

ServerInfo.sin\_port = htons(PORT); // порт сервера

exodus = connect(ClientSock, (SOCKADDR\*)&ServerInfo, sizeof(ServerInfo));

if (exodus == SOCKET\_ERROR) {

printf("connect failed: %d\n", WSAGetLastError());

closesocket(ClientSock);

WSACleanup();

return 1;

}

// Отправляем и получаем данные от сервера

printf("Connected to server on port %d\n", PORT);

char\* buffer = (char\*)malloc(BUFFER\_SIZE \* sizeof(char));

int bytes\_sent, bytes\_received;

while (1) {

// Вводим запрос

printf("Enter your query: ");

fgets(buffer, BUFFER\_SIZE, stdin);

buffer[strcspn(buffer, "\n")] = '\0'; // удаляем символ перевода строки

// Отправляем запрос

bytes\_sent = send(ClientSock, buffer, strlen(buffer), 0);

if (bytes\_sent == SOCKET\_ERROR) {

printf("send failed: %d\n", WSAGetLastError());

break;

}

// Получаем ответ

bytes\_received = recv(ClientSock, buffer, BUFFER\_SIZE, 0);

if (bytes\_received == SOCKET\_ERROR) {

printf("recv failed: %d\n", WSAGetLastError());

break;

}

buffer[bytes\_received] = '\0';

// Выводим ответ

printf("Server response: %s\n", buffer);

}

// Закрываем сокет клиента и очищаем библиотеку Winsock

free(buffer);

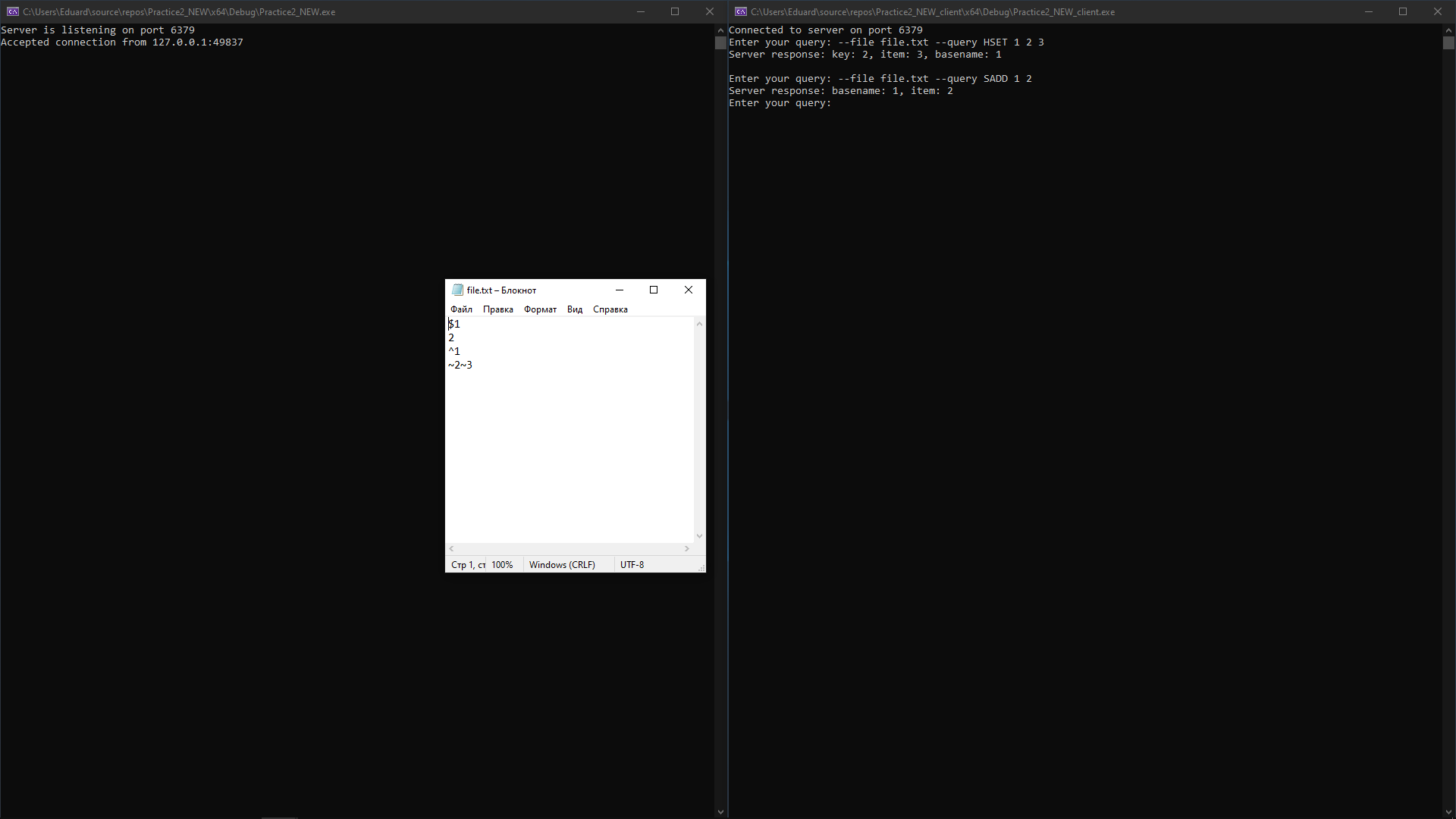
closesocket(ClientSock);

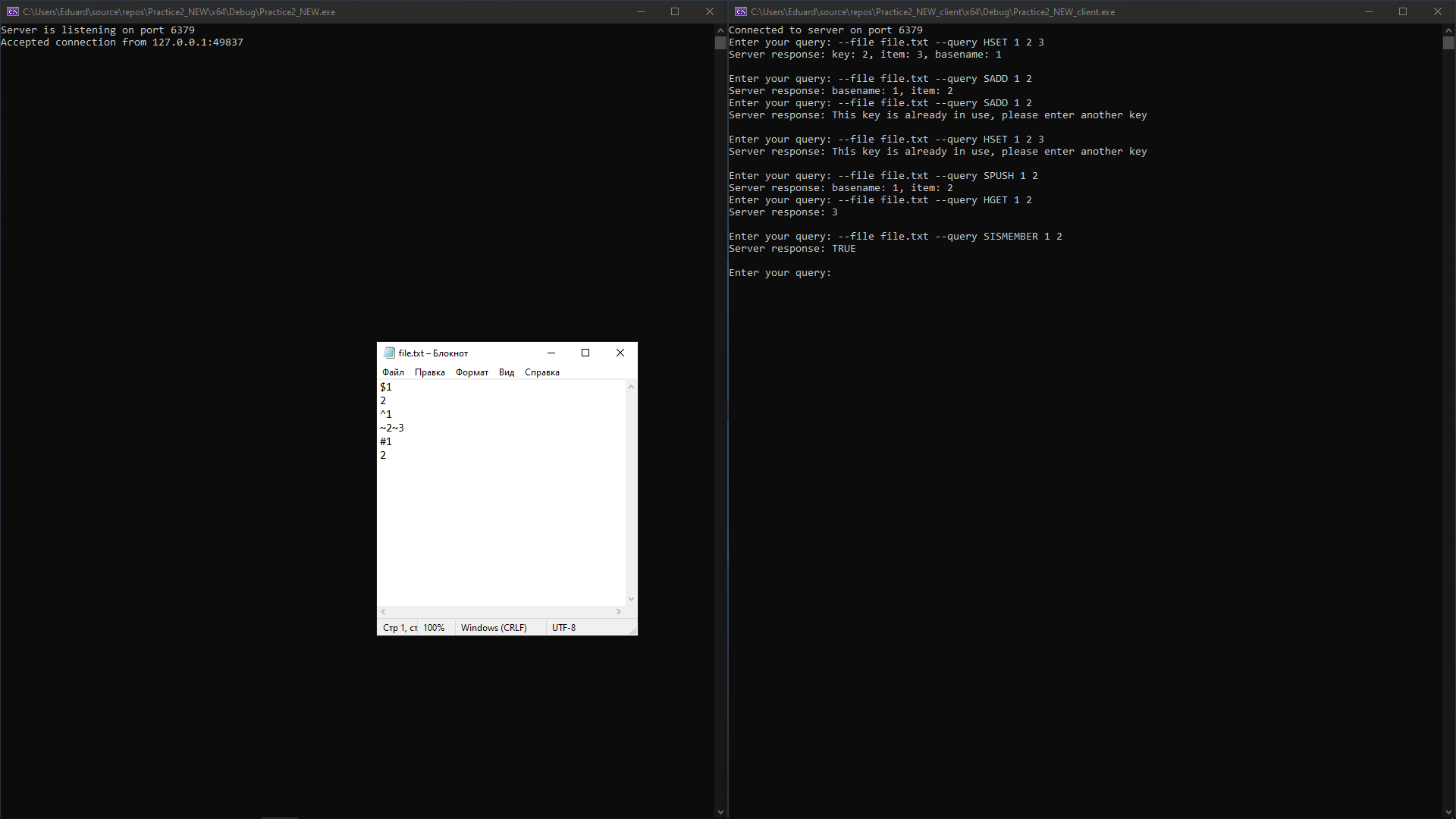
WSACleanup();

return 0;

}

**Результат работы программы**





**Вывод**: Система управления СУБД работает исправно, для подтверждения этого были предоставлены скриншоты работы выше.