МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ образовательное учреждениевысшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации



**ОТЧЁТ по ПРАКТИЧЕСКОЙ работе №3**

**по дисциплине «ПРОГРАММИРОВАНИЕ»**

**«Разработка сервиса сокращения ссылок»**

|  |  |
| --- | --- |
| Факультет: АВТФ  Группа: АБс-222  Студент: Гатауллин Д.Р. | Преподаватель: Архипова А.Б. |

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

**Цели и задачи работы**: Изучение тем Сеть, ООП, Web.

**Задание к работе**: Требуется реализовать сервис для сокращения ссылок. Сервис должен принимать запросы по http протоколу.

Сервис должен иметь две конечные точки:

1. Сокращение ссылки

URL: /

Метод: POST

Запрос содержит ссылку которую нужно сократить.

Ответ содержит сокращенную ссылку.

2. Переход по сокращённой ссылке

URL: /<сокращение>

Метод: GET

Ответ должен перенаправлять пользователя на исходную

страницу (см. перенаправления)

Для хранения сокращенных ссылок используем СУБД из практики

2.

2. Реализовать клиента для СУБД на том языке, на которым вы

пишите сервис сокращения ссылок.

Пример аналогичного сервиса: <https://clck.ru>

**Текст программы**

==========================================================

С

==========================================================

main.c

==========================================================

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "stack.h"

#include "set.h"

#include "queue.h"

#include "table.h"

#include <winsock2.h>

#include <ws2tcpip.h>

#include <windows.h>

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

#define PORT 6379 // Определение порта для сервера

#define BACKLOG 10 // Определение максимального количества ожидающих подключений

#define BUFFER\_SIZE 104857600 // Определение размера буфера

HANDLE mutex; // Объявление мьютекса

DWORD WINAPI handle\_client(LPVOID lpParam); // Объявление функции обработки клиента

int main() {

WSADATA wsa\_data;

int result = WSAStartup(MAKEWORD(2, 2), &wsa\_data); // Инициализация библиотеки сокетов

SOCKET listen\_socket = socket(AF\_INET, SOCK\_STREAM, IPPROTO\_TCP); // Создание сокета для прослушивания

struct sockaddr\_in server\_address; // Структура для хранения адреса сервера

server\_address.sin\_family = AF\_INET;

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

server\_address.sin\_port = htons(PORT);

result = bind(listen\_socket, (struct sockaddr\*)&server\_address, sizeof(server\_address)); // Привязка сокета к адресу

result = listen(listen\_socket, BACKLOG); // Начало прослушивания сокета

printf("The server is running and waiting for connections on port %d\n", PORT);

mutex = CreateMutex(NULL, FALSE, NULL); // Создание мьютекса

while (1) {

SOCKET client\_socket = accept(listen\_socket, NULL, NULL); // Принятие подключения от клиента

CreateThread(NULL, 0, handle\_client, (LPVOID)client\_socket, 0, NULL); // Создание потока для обработки клиента

}

closesocket(listen\_socket); // Закрытие сокета прослушивания

WSACleanup(); // Очистка библиотеки сокетов

return 0;

}

DWORD WINAPI handle\_client(LPVOID lpParam) {

WaitForSingleObject(mutex, INFINITE); // Получение мьютекса

SOCKET client\_socket = (SOCKET)lpParam; // Получение сокета клиента

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

int bytes\_received;

// Получение данных от клиента, пока клиент не закроет подключение

while ((bytes\_received = recv(client\_socket, buffer, BUFFER\_SIZE, 0)) > 0) {

buffer[bytes\_received] = '\0'; // Добавление нуль-терминатора

// Удаление символов возврата каретки и новой строки

for (int i = 0; i < bytes\_received; ++i) {

if (buffer[i] == '\r' || buffer[i] == '\n') {

buffer[i] = '\0';

break;

}

}

char\*\* argv = NULL;

int argc = 0;

char\* token = strtok(buffer, " "); // Разбиение полученных данных на токены

while (token != NULL) {

char\*\* temp = realloc(argv, sizeof(char\*) \* (argc + 1));

argv = temp;

argv[argc++] = token;

token = strtok(NULL, " ");

}

// Обработка полученных данных и выполнение соответствующих операций

char\* filename = NULL; // Имя файла

char\* query = NULL;

char\* key = NULL; // Ключ (Объект)

char\* basename = NULL; // Имя БД

char\* item = NULL; // Объект

int temp; // Переменная, отвечающая за номер аргумента

char\* result = NULL;

// Проверка количества аргументов

if (argc < 4 || argc > 7 || argv[4] == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

// Цикл по аргументам

for (int i = 0; i < argc; i++) {

if (strcmp(argv[i], "--file") == 0 && i + 1 < argc) {

filename = argv[i + 1];

}

else if (strcmp(argv[i], "--query") == 0 && i + 1 < argc) {

query = argv[i + 1];

temp = i + 1;

basename = argv[i + 2];

if (i + 5 > argc) key = argv[i + 3];

else {

item = argv[i + 3];

key = argv[i + 4];

// Проверка наличия ключа и элемента

if (key == NULL || item == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

}

}

}

int pos1 = 0; // Переменная, отвечающая за позицию начала строки.

int pos2 = 0; // Переменная, отвечающая за позицию конца строки.

int status = 0; // Переменная-переключатель.

// Обработка каждой команды

if (filename != NULL && query != NULL) {

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

if (!file) {

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

}

if (strcmp(argv[temp], "SPUSH") == 0) {

if (key == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

Stack\* stack = loadFromFileStack(filename, basename, &pos1, &pos2, &status);

if (stack == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

SPUSH(stack, key);

result = malloc(strlen(key) + 5);

sprintf(result, "-> %s\n", key);

if (status == 1) status = 0;

fclose(file);

saveToFileStack(stack, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "SPOP") == 0) {

Stack\* stack = loadFromFileStack(filename, basename, &pos1, &pos2, &status);

if (stack == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(40);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

char\* element = SPOP(stack);

result = malloc(strlen(element) + 5);

sprintf(result, "-> %s\n", element);

if (status == 2) status = 0;

fclose(file);

saveToFileStack(stack, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "SADD") == 0) {

if (key == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

Set\* set = loadFromFileSet(filename, basename, &pos1, &pos2, &status);

if (set == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

SADD(set, key);

result = malloc(strlen(key) + 5);

sprintf(result, "-> %s\n", key);

if (status == 1) status = 0;

fclose(file);

saveToFileSet(set, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "SREM") == 0) {

if (key == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

Set\* set = loadFromFileSet(filename, basename, &pos1, &pos2, &status);

if (set == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

SREM(set, key);

result = malloc(strlen(key) + 5);

sprintf(result, "-> %s\n", key);

if (status == 2) status = 0;

fclose(file);

saveToFileSet(set, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "SISMEMBER") == 0) {

if (key == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

Set\* set = loadFromFileSet(filename, basename, &pos1, &pos2, &status);

if (set == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

result = malloc(100);

if (SISMEMBER(set, key)) sprintf(result, "-> True\n");

else sprintf(result, "-> False\n");

fclose(file);

}

}

}

if (strcmp(argv[temp], "QPUSH") == 0) {

if (key == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

Queue\* queue = loadFromFileQueue(filename, basename, &pos1, &pos2, &status);

if (queue == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

QPUSH(queue, key);

result = malloc(strlen(key) + 5);

sprintf(result, "-> %s\n", key);

if (status == 1) status = 0;

fclose(file);

saveToFileQueue(queue, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "QPOP") == 0) {

Queue\* queue = loadFromFileQueue(filename, basename, &pos1, &pos2, &status);

if (queue == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

char\* element = QPOP(queue);

result = malloc(strlen(element) + 5);

sprintf(result, "-> %s\n", element);

if (status == 2) status = 0;

fclose(file);

saveToFileQueue(queue, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "HSET") == 0) {

if (key == NULL || item == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

HashTable\* hashtable = loadFromFileTable(filename, basename, &pos1, &pos2, &status);

if (hashtable == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

HSET(hashtable, key, item);

result = malloc(strlen(item) + strlen(key) + 20);

sprintf(result, "-> %s %s\n", item, key);

if (status == 1) status = 0;

fclose(file);

saveToFileTable(hashtable, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "HDEL") == 0) {

if (key == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

HashTable\* hashtable = loadFromFileTable(filename, basename, &pos1, &pos2, &status);

if (hashtable == NULL) {

result = malloc(100);

sprintf(result, "Error when opening a file!\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Such a database, alas, does not exist!\n");

fclose(file);

}

else {

HDEL(hashtable, key);

result = malloc(strlen(key) + 5);

sprintf(result, "-> %s\n", key);

if (status == 2) status = 0;

fclose(file);

saveToFileTable(hashtable, filename, basename, &pos1, &pos2, &status);

}

}

}

if (strcmp(argv[temp], "HGET") == 0) {

if (key == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

goto skip;

}

HashTable\* hashtable = loadFromFileTable(filename, basename, &pos1, &pos2, &status);

if (hashtable == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

fclose(file);

}

else {

if (pos1 + pos2 == 0) {

result = malloc(100);

sprintf(result, "Error.\n");

fclose(file);

}

else {

if (HGET(hashtable, key) != NULL) {

result = malloc(strlen(HGET(hashtable, key)) + 15);

sprintf(result, "%s\n", HGET(hashtable, key));

}

else {

result = malloc(100);

sprintf(result, "-> False\n");

}

fclose(file);

}

}

}

}

else {

result = malloc(100);

sprintf(result, "Error.\n");

}

skip: {

if (result == NULL) {

result = malloc(100);

sprintf(result, "Error.\n");

}

int bytes\_sent = send(client\_socket, result, strlen(result), 0); // Отправка результата клиенту

}

free(argv);

free(result);

}

free(buffer);

closesocket(client\_socket); // Закрытие сокета клиента

ReleaseMutex(mutex); // Освобождение мьютекса

}

==========================================================

queue.c

==========================================================

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "queue.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Функция подсчета слов в файле

int countWordsInFileQueue(const char\* filename) {

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

int count = 0;

char word[10000];

while (fscanf(file, "%s", word) != EOF) {

count++;

}

fclose(file);

return count;

}

// Функция инициализации очереди

Queue\* initQueue() {

Queue\* queue = (Queue\*)malloc(sizeof(Queue));

queue->front = NULL; // Установка указателя на начало очереди в NULL

queue->rear = NULL; // Установка указателя на конец очереди в NULL

return queue; // Возврат указателя на очередь

}

// Функция добавления элемента в очередь

void QPUSH(Queue\* queue, char\* element) {

// Новый узел

NodeQueue\* newNode = (NodeQueue\*)malloc(sizeof(NodeQueue));

newNode->element = \_strdup(element); // Копирование элемента в узел

newNode->next = NULL; // Установка указателя на следующий узел в NULL

if (queue->front == NULL) { // Если очередь пуста

queue->front = newNode; // Установка указателя на начало очереди на новый узел

queue->rear = newNode; // Установка указателя на конец очереди на новый узел

}

else { // Если очередь не пуста

queue->rear->next = newNode; // Добавление нового узла в конец очереди

queue->rear = newNode; // Установка указателя на конец очереди на новый узел

}

}

// Функция удаления элемента из очереди

char\* QPOP(Queue\* queue) {

// Проверка пустоты очереди

if (queue->front == NULL) {

return ("The queue is empty");

}

NodeQueue\* poppedNode = queue->front; // Указатель на удаляемый узел

char\* element = poppedNode->element; // Указатель на удаляемый элемент

queue->front = poppedNode->next; // Сдвиг указателя на начало очереди на следующий узел

if (queue->front == NULL) { // Если очередь стала пустой

queue->rear = NULL; // Установка указателя на конец очереди в NULL

}

free(poppedNode); // Освобождение памяти от удаляемого узла

return element; // Возврат указателя на удаляемый элемент

}

// Функция сохранения изменений в файл

void saveToFileQueue(Queue\* queue, const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

FILE\* tempFile = fopen("temp.data", "w");

int ch; // Символ, который требуется записать

fseek(file, 0, SEEK\_SET); // Установка курсора в начало

fseek(tempFile, 0, SEEK\_SET); // Установка курсора в начало

while ((ch = fgetc(file)) != EOF) { // Цикл, пока не выдаст ошибку при записи

fputc(ch, tempFile); // Запись символа в поток

if (ftell(tempFile) == \*pos1 - 2 && \*status == 2) // Если пустая БД, то добавляем табуляцию перед элементом

fprintf(tempFile, "\t%s", queue->front->element);

else if (ftell(tempFile) == \*pos1) {

NodeQueue\* currentNode = queue->front; // Ставим указатель на начало

// Пока не закончатся добавляем элементы

while (currentNode != NULL) {

if (currentNode->next == NULL) // Если нет следующего элемента, то добавляем строку

fprintf(tempFile, "%s\n", currentNode->element);

else // Иначе добавляем после элемента табуляцию

fprintf(tempFile, "%s\t", currentNode->element);

currentNode = currentNode->next; // Переход к следующему элементу

}

// Добавление новую строку, если БД пуста

if (\*status == 1) {

fseek(tempFile, \*pos1 - 1, SEEK\_SET);

fprintf(tempFile, "\n");

}

fseek(file, \*pos2, SEEK\_SET);

}

}

NodeQueue\* currentNode = queue->front;

while (currentNode != NULL) {

NodeQueue\* nextNode = currentNode->next;

free(currentNode->element);

free(currentNode);

currentNode = nextNode;

}

free(queue);

fclose(file);

fclose(tempFile);

remove(filename);

rename("temp.data", filename);

}

// Функция загрузки БД в структуру

Queue\* loadFromFileQueue(const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

// Проверка наличия файла

if (file == NULL) {

return NULL;

}

int num\_lines = countWordsInFileQueue(filename); // Переменная, отвечающая за количество слов

char\*\* line = malloc(num\_lines \* sizeof(char\*));

for (int i = 0; i < num\_lines; i++) line[i] = malloc(10000 \* sizeof(char));

Queue\* queue = initQueue();

int tempory = 0; // Переменная-переключатель

int pos3 = 0; // Временная переменная, которая запоминает исходную позицию курсора

int temp1 = 0; // Переменная, отвечающая за номер 1 элемента в БД

int temp2 = 0; // Переменная, отвечающая за номер 2 элемента в БД

char c = '1'; // Полученный символ

// Проходимся по всем словам

for (int i = 0; i < num\_lines; ++i) {

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

c = getc(file);

pos3 = ftell(file);

if (!strcmp(line[i], basename)) { // Если наша БД, то делаем с ней операции

fseek(file, -3 - strlen(line[i]), SEEK\_CUR); // Временно переносим курсор на предыдущии позиции

if (getc(file) == '\n' || i == 0) {

fseek(file, pos3, SEEK\_SET); // Возврат курсора

tempory = 1;

\*pos1 = ftell(file);

temp1 = i + 1;

}

else fseek(file, pos3, SEEK\_SET); // Возврат курсора

}

if (c == '\n' && tempory == 1) { // Конец нашей БД

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

}

if (feof(file))

break;

}

if (temp1 == temp2)

\*status = 1; // 1 элемент

if (temp1 == temp2 + 1)

\*status = 2; // 2 или более элемента

// Добавление в структуру элементов

while (temp1 < temp2 + 1) {

QPUSH(queue, line[temp1]);

temp1++;

}

fclose(file);

for (int i = 0; i < num\_lines; i++) {

free(line[i]);

}

free(line);

return queue;

}

==========================================================

queue.h

==========================================================

#ifndef QUEUE\_H

#define QUEUE\_H

typedef struct NodeQueue {

char\* element;

struct NodeQueue\* next;

} NodeQueue;

typedef struct Queue {

NodeQueue\* front;

NodeQueue\* rear;

} Queue;

Queue\* initQueue();

void QPUSH(Queue\* queue, char\* element);

char\* QPOP(Queue\* queue);

void saveToFileQueue(Queue\* queue, const char\* filename, const char\* basename, int \*pos1, int \*pos2, int \*status);

Queue\* loadFromFileQueue(const char\* filename, const char\* basename, int \*pos1, int \*pos2, int \*status);

#endif

==========================================================

set.c

==========================================================

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "set.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100000

// Функция подсчета слов в файле

int countWordsInFileSet(const char\* filename) {

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

int count = 0; // Счетчик слов

char word[10000]; // Массив для хранения слов

while (fscanf(file, "%s", word) != EOF) {

count++;

}

fclose(file);

return count;

}

// Функция для инициализации множества

Set\* initSet() {

Set\* set = (Set\*)malloc(sizeof(Set));

set->head = NULL; // Инициализация головы множества

set->size = 0; // Инициализация размера множества

set->tableSize = MAX\_SIZE; // Инициализация размера хеш-таблицы

set->hashTable = (Node\*\*)malloc(MAX\_SIZE \* sizeof(Node\*));

set->emptySlots = (int\*)malloc(MAX\_SIZE \* sizeof(int));

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

set->hashTable[i] = NULL; // Инициализация элемента хеш-таблицы

set->emptySlots[i] = 1; // Инициализация элемента массива пустых слотов

}

return set; // Возврат указателя на множество

}

// Функция для вычисления первого хеша

int calculateHashS(const char\* element) {

int hash = 0;

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

hash = 31 \* hash + element[i];

}

return abs(hash) % MAX\_SIZE;

}

// Функция для вычисления второго хеша

int calculateHash2S(const char\* element) {

int hash = 0;

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

hash = 17 \* hash + element[i];

}

return abs(hash) % MAX\_SIZE;

}

// Функция для добавления элемента в множество

void SADD(Set\* set, char\* element) {

int hash = calculateHashS(element); // Вычисление первого хеша элемента

int step = calculateHash2S(element); // Вычисление второго хеша элемента

// Объединение хешов

while (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) != 0) {

hash = (hash + step) % MAX\_SIZE;

}

// Проверка повтора элемента

if (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) == 0) {

return;

}

// Новый узел

Node\* newNode = (Node\*)malloc(sizeof(Node));

newNode->element = \_strdup(element); // Копирование элемента

newNode->hash = hash; // Сохранение хеша элемента

newNode->next = set->head; // Установка следующего узла после нового узла

if (set->head != NULL) { // Если голова множества не пуста

set->head->prev = newNode; // Установка предыдущего узла

}

set->head = newNode; // Обновление головы списка

set->hashTable[hash] = newNode; // Вставка узла в хеш-таблицу

set->size++; // Увеличение размера множества

}

// Функция для удаления элемента из множества

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

int hash = calculateHashS(element); // Вычисление первого хеша элемента

int step = calculateHash2S(element); // Вычисление второго хеша элемента

// Объединение хешов

while (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) != 0) {

hash = (hash + step) % MAX\_SIZE;

}

// Если найден элемент

if (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) == 0) {

Node\* nodeToRemove = set->hashTable[hash]; // Получение узла для удаления

if (nodeToRemove == set->head) { // Если узел для удаления является головой множества

set->head = nodeToRemove->next; // Установка следующего узла в качестве головы множества

}

else {

if (nodeToRemove->prev != NULL) { // Если у узла для удаления есть предыдущий узел

nodeToRemove->prev->next = nodeToRemove->next; // Установка следующего узла после предыдущего узла

}

}

if (nodeToRemove->next != NULL) { // Если у узла для удаления есть следующий узел

nodeToRemove->next->prev = nodeToRemove->prev; // Установка предыдущего узла перед следующим узлом

}

free(nodeToRemove->element); // Освобождение памяти, занятой элементом

free(nodeToRemove); // Освобождение памяти, занятой узлом

set->hashTable[hash] = NULL; // Удаление узла из хеш-таблицы

set->size--; // Уменьшение размера множества на 1

return;

}

}

// Функция для проверки существования элемента

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

int hash = calculateHashS(element); // Вычисление первого хеша элемента

int step = calculateHash2S(element); // Вычисление второго хеша элемента

// Объединение хешов

while (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) != 0) {

hash = (hash + step) % MAX\_SIZE;

}

// Если найден элемент

if (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) == 0) {

return 1;

}

return 0;

}

// Функция сохранения изменений в файл

void saveToFileSet(Set\* set, const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

FILE\* tempFile = fopen("temp.data", "w");

int ch; // Символ, который требуется записать

fseek(file, 0, SEEK\_SET); // Установка курсора в начало

fseek(tempFile, 0, SEEK\_SET); // Установка курсора в начало

char\*\* elements = (char\*\*)malloc(set->size \* sizeof(char\*));

Node\* current = set->head; // Ставим указатель на голову

int i = 0; // Переменная, отвечающая за порядок

// Добавление элементов в структуру

while (current != NULL) {

elements[i] = current->element;

current = current->next;

i++;

}

while ((ch = fgetc(file)) != EOF) { // Цикл, пока не выдаст ошибку при записи

fputc(ch, tempFile); // Запись символа в поток

if (ftell(tempFile) == \*pos1 - 2 && \*status == 2) // Если пустая БД, то добавляем табуляцию перед элементом

fprintf(tempFile, "\t%s", set->head->element);

else if (ftell(tempFile) == \*pos1) {

// Пока не закончатся добавляем элементы

for (int j = set->size - 1; j >= 0; j--) {

fprintf(tempFile, "%s", elements[j]);

if (j > 0) // Если есть следующий элемент, то добавляем табуляцию после элемента

fprintf(tempFile, "\t");

else // Иначе добавляем новую строку

fprintf(tempFile, "\n");

}

// Добавление новую строку, если БД пуста

if (\*status == 1) {

fseek(tempFile, \*pos1 - 1, SEEK\_SET);

fprintf(tempFile, "\n");

}

fseek(file, \*pos2, SEEK\_SET);

}

}

free(elements);

free(set->hashTable);

free(set->emptySlots);

free(set);

fclose(file);

fclose(tempFile);

remove(filename);

rename("temp.data", filename);

}

// Функция загрузки БД в структуру

Set\* loadFromFileSet(const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

// Проверка наличия файла

if (file == NULL) {

return NULL;

}

int num\_lines = countWordsInFileSet(filename); // Переменная, отвечающая за количество слов

char\*\* line = malloc(num\_lines \* sizeof(char\*));

for (int i = 0; i < num\_lines; i++) line[i] = malloc(10000 \* sizeof(char));

Set\* set = initSet();

int tempory = 0; // Переменная-переключатель

int pos3 = 0; // Временная переменная, которая запоминает исходную позицию курсора

int temp1 = 0; // Переменная, отвечающая за номер 1 элемента в БД

int temp2 = 0; // Переменная, отвечающая за номер 2 элемента в БД

char c = '1'; // Полученный символ

// Проходимся по всем словам

for (int i = 0; i < num\_lines; ++i) {

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

c = getc(file);

pos3 = ftell(file);

if (!strcmp(line[i], basename)) { // Если наша БД, то делаем с ней операции

fseek(file, -3 - strlen(line[i]), SEEK\_CUR); // Временно переносим курсор на предыдущии позиции

if (getc(file) == '\n' || i == 0) {

fseek(file, pos3, SEEK\_SET); // Возврат курсора

tempory = 1;

\*pos1 = ftell(file);

temp1 = i + 1;

}

else fseek(file, pos3, SEEK\_SET); // Возврат курсора

}

if (c == '\n' && tempory == 1) { // Конец нашей БД

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

}

if (feof(file))

break;

}

if (temp1 == temp2)

\*status = 1; // 1 элемент

if (temp1 == temp2 + 1)

\*status = 2; // 2 или более элемента

// Добавление в структуру элементов

while (temp1 < temp2 + 1) {

SADD(set, line[temp1]);

temp1++;

}

fclose(file);

for (int i = 0; i < num\_lines; i++) {

free(line[i]);

}

free(line);

return set;

}

==========================================================

set.h

==========================================================

#ifndef SET\_H

#define SET\_H

typedef struct Node {

char\* element;

int hash;

struct Node\* next;

struct Node\* prev;

} Node;

typedef struct Set {

Node\* head;

int size;

Node\*\* hashTable;

int tableSize;

int\* emptySlots;

} Set;

Set\* initSet();

void SADD(Set\* set, char\* element);

void SREM(Set\* set, char\* element);

int SISMEMBER(Set\* set, char\* element);

void saveToFileSet(Set\* set, const char\* filename, const char\* basename, int \*pos1, int \*pos2, int \*status);

Set\* loadFromFileSet(const char\* filename, const char\* basename, int \*pos1, int \*pos2, int \*status);

#endif

==========================================================

stack.c

==========================================================

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "stack.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

// Функция подсчета слов в файле

int countWordsInFileStack(const char\* filename) {

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

int count = 0;

char word[10000];

while (fscanf(file, "%s", word) != EOF) {

count++;

}

fclose(file);

return count;

}

// Функция для инициализации стека

Stack\* initStack() {

Stack\* stack = (Stack\*)malloc(sizeof(Stack));

stack->top = NULL;

return stack;

}

// Функция для добавления элемента в стек

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

NodeStack\* newNode = (NodeStack\*)malloc(sizeof(NodeStack)); // Выделение памяти под новый узел

newNode->element = \_strdup(element); // Копирование элемента в узел

newNode->next = stack->top; // Установка следующего узла после нового узла

stack->top = newNode; // Установка нового узла в качестве вершины стека

}

// Функция для удаления элемента из стека

char\* SPOP(Stack\* stack) {

// Проверка пустоты стека

if (stack->top == NULL) {

return ("The stack is empty");

}

NodeStack\* poppedNode = stack->top; // Получение узла для удаления

stack->top = poppedNode->next; // Установка следующего узла в качестве вершины стека

char\* element = poppedNode->element; // Получение элемента из узла

free(poppedNode); // Освобождение памяти, занятой узлом

return element; // Возврат элемента

}

// Функция сохранения изменений в файл

void saveToFileStack(Stack\* stack, const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

FILE\* tempFile = fopen("temp.data", "w");

int ch; // Символ, который требуется записать

fseek(file, 0, SEEK\_SET); // Установка курсора в начало

fseek(tempFile, 0, SEEK\_SET); // Установка курсора в начало

while ((ch = fgetc(file)) != EOF) { // Цикл, пока не выдаст ошибку при записи

fputc(ch, tempFile); // Запись символа в поток

if (ftell(tempFile) == \*pos1 - 2 && \*status == 2) // Если пустая БД, то добавляем табуляцию перед элементом

fprintf(tempFile, "\t%s", stack->top->element);

else if (ftell(tempFile) == \*pos1) {

NodeStack\* currentNode = stack->top; // Ставим указатель на вершину стека

NodeStack\* prevNode = NULL; // Временная переменная-указатель

// Цикл для перевёртывания стека

while (currentNode != NULL) {

NodeStack\* nextNode = currentNode->next;

currentNode->next = prevNode;

prevNode = currentNode;

currentNode = nextNode;

}

currentNode = prevNode;

while (currentNode != NULL) { // Цикл пока не закончатся элементы

if (currentNode->next == NULL)

fprintf(tempFile, "%s\n", currentNode->element); // Если нет следующего элемента, то добавляем строку

else // Иначе добавляем после элемента табуляцию

fprintf(tempFile, "%s\t", currentNode->element);

currentNode = currentNode->next;

}

// Добавление новую строку, если БД пуста

if (\*status == 1) {

fseek(tempFile, \*pos1 - 1, SEEK\_SET);

fprintf(tempFile, "\n");

}

fseek(file, \*pos2, SEEK\_SET);

}

}

free(stack->top);

free(stack);

fclose(file);

fclose(tempFile);

remove(filename);

rename("temp.data", filename);

}

// Функция загрузки БД в структуру

Stack\* loadFromFileStack(const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

// Проверка наличия файла

if (file == NULL) {

return NULL;

}

int num\_lines = countWordsInFileStack(filename); // Переменная, отвечающая за количество слов

char\*\* line = malloc(num\_lines \* sizeof(char\*));

for (int i = 0; i < num\_lines; i++) line[i] = malloc(10000 \* sizeof(char));

Stack\* stack = initStack();

int tempory = 0; // Переменная-переключатель

int pos3 = 0; // Временная переменная, которая запоминает исходную позицию курсора

int temp1 = 0; // Переменная, отвечающая за номер 1 элемента в БД

int temp2 = 0; // Переменная, отвечающая за номер 2 элемента в БД

char c = '1'; // Полученный символ

// Проходимся по всем словам

for (int i = 0; i < num\_lines; ++i) {

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

c = getc(file);

pos3 = ftell(file);

if (!strcmp(line[i], basename)) { // Если наша БД, то делаем с ней операции

fseek(file, -3 - strlen(line[i]), SEEK\_CUR); // Временно переносим курсор на предыдущии позиции

if (getc(file) == '\n' || i == 0) {

fseek(file, pos3, SEEK\_SET); // Возврат курсора

tempory = 1;

\*pos1 = ftell(file);

temp1 = i + 1;

}

else fseek(file, pos3, SEEK\_SET); // Возврат курсора

}

if (c == '\n' && tempory == 1) { // Конец нашей БД

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

}

if (feof(file))

break;

}

if (temp1 == temp2)

\*status = 1; // 1 элемент

if (temp1 == temp2 + 1)

\*status = 2; // 2 или более элемента

// Добавление в структуру элементов

while (temp1 < temp2 + 1) {

SPUSH(stack, line[temp1]);

temp1++;

}

fclose(file);

for (int i = 0; i < num\_lines; i++) {

free(line[i]);

}

free(line);

return stack;

}

==========================================================

stack.h

==========================================================

#ifndef STACK\_H

#define STACK\_H

typedef struct NodeStack {

char\* element;

struct NodeStack\* next;

} NodeStack;

typedef struct Stack {

NodeStack\* top;

} Stack;

Stack\* initStack();

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

char\* SPOP(Stack\* stack);

void saveToFileStack(Stack\* stack, const char\* filename, const char\* basename, int \*pos1, int \*pos2, int \*status);

Stack\* loadFromFileStack(const char\* filename, const char\* basename, int \*pos1, int \*pos2, int \*status);

#endif

==========================================================

table.c

==========================================================

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "table.h"

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#define MAX\_SIZE 100000

// Функция подсчета слов в файле

int countWordsInFileTable(const char\* filename) {

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

int count = 0;

char word[10000];

while (fscanf(file, "%s", word) != EOF) {

count++;

}

fclose(file);

return count;

}

// Функция для инициализации хеш-таблицы

HashTable\* initHashTable() {

HashTable\* ht = (HashTable\*)malloc(sizeof(HashTable));

ht->nodes = (NodeHashTable\*\*)malloc(MAX\_SIZE \* sizeof(NodeHashTable\*));

ht->count = 0;

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

ht->nodes[i] = NULL;

}

return ht;

}

// Функция для вычисления хеша

int calculateHashT(const char\* element) {

int hash = 0;

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

hash = 31 \* hash + element[i];

}

return abs(hash) % MAX\_SIZE;

}

// Функция для добавления элемента в хеш-таблицу

void HSET(HashTable\* ht, char\* key, char\* value) {

int hash = calculateHashT(key);

// Создаем новый узел для хранения ключа и значения

NodeHashTable\* newNode = (NodeHashTable\*)malloc(sizeof(NodeHashTable));

newNode->key = \_strdup(key);

newNode->element = \_strdup(value);

newNode->next = NULL; // Устанавливаем указатель на следующий узел как NULL

newNode->prev = NULL; // Устанавливаем указатель на предыдущий узел как NULL

// Обработка коллизий и проверка на дубликаты ключей

NodeHashTable\* current = ht->nodes[hash]; // Получаем узел по хеш-ключу

while (current != NULL) {

if (strcmp(current->key, key) == 0) { // Если ключ уже существует

// Освобождаем память нового узла

free(newNode->key);

free(newNode->element);

free(newNode);

return;

}

if (current->next == NULL) { // Если достигли конца цепочки

break;

}

current = current->next; // Переходим к следующему узлу

}

// Добавление нового узла

if (current == NULL) { // Если цепочка пуста

ht->nodes[hash] = newNode; // Устанавливаем новый узел как начало цепочки

}

else {

current->next = newNode; // Добавляем новый узел в конец цепочки

newNode->prev = current; // Устанавливаем предыдущий узел для нового узла

}

ht->count++;

}

// Функция для получения элемента из хеш-таблицы

char\* HGET(HashTable\* ht, const char\* key) {

int hash = calculateHashT(key);

NodeHashTable\* current = ht->nodes[hash]; // Получаем узел по хеш-ключу

while (current != NULL) { // Перебираем узлы в цепочке

if (strcmp(current->key, key) == 0) { // Если ключ совпадает

return current->element;

}

current = current->next; // Переходим к следующему узлу

}

return NULL;

}

// Функция для удаления элемента из хеш-таблицы

void HDEL(HashTable\* ht, const char\* key) {

int hash = calculateHashT(key);

NodeHashTable\* current = ht->nodes[hash]; // Получаем узел по хеш-ключу

NodeHashTable\* nodeToRemove = NULL;

while (current != NULL) { // Перебираем узлы в цепочке

if (strcmp(current->key, key) == 0) { // Если ключ совпадает

nodeToRemove = current; // Устанавливаем узел для удаления

break;

}

current = current->next; // Переходим к следующему узлу

}

if (nodeToRemove != NULL) {

if (nodeToRemove->prev != NULL) { // Если у узла есть предыдущий узел

nodeToRemove->prev->next = nodeToRemove->next; // Удаляем узел из цепочки

}

else {

ht->nodes[hash] = nodeToRemove->next; // Устанавливаем следующий узел как начало цепочки

}

if (nodeToRemove->next != NULL) { // Если у узла есть следующий узел

nodeToRemove->next->prev = nodeToRemove->prev; // Устанавливаем предыдущий узел для следующего узла

}

// Освобождаем память удаляемого узла

free(nodeToRemove->key);

free(nodeToRemove->element);

free(nodeToRemove);

ht->count--;

}

}

void saveToFileTable(HashTable\* hashtable, const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

FILE\* tempFile = fopen("temp.data", "w");

int ch;

int count = 0;

fseek(file, 0, SEEK\_SET);

fseek(tempFile, 0, SEEK\_SET);

while ((ch = fgetc(file)) != EOF) {

fputc(ch, tempFile);

if (ftell(tempFile) == \*pos1 || (ftell(tempFile) == \*pos1 - 2 && \*status == 2)) {

// Если пустая БД, то добавляем табуляцию перед элементом с ключом

if (hashtable->count == 1 && \*status == 2) {

fprintf(tempFile, "\t");

}

// Проходимся по всем элементам

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

NodeHashTable\* currentNode = hashtable->nodes[i];

while (currentNode != NULL) {

fprintf(tempFile, "%s\t%s", currentNode->element, currentNode->key);

count++;

if (count != hashtable->count) {

fprintf(tempFile, "\t"); // Добавляем табуляцию после элемента с ключом

}

else {

fprintf(tempFile, "\n"); // Если нет следующего элемента с ключом, то добавляем строку

break;

}

currentNode = currentNode->next;

}

}

// Добавление новую строку, если БД пуста

if (\*status == 1) {

fseek(tempFile, \*pos1 - 1, SEEK\_SET);

fprintf(tempFile, "\n");

}

fseek(file, \*pos2, SEEK\_SET);

}

}

fclose(file);

fclose(tempFile);

remove(filename);

rename("temp.data", filename);

}

// Функция загрузки БД в структуру

HashTable\* loadFromFileTable(const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status) {

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

// Проверка наличия файла

if (file == NULL) {

return NULL;

}

int num\_lines = countWordsInFileTable(filename); // Переменная, отвечающая за количество слов

char\*\* line = malloc(num\_lines \* sizeof(char\*));

for (int i = 0; i < num\_lines; i++) line[i] = malloc(10000 \* sizeof(char));

HashTable\* hashtable = initHashTable();

int tempory = 0; // Переменная-переключатель

int tempory2 = 0; // Переменная, временно хранящие положение символа

int temp1 = 0; // Переменная, отвечающая за номер 1 элемента в БД

int temp2 = 0; // Переменная, отвечающая за номер 2 элемента в БД

char c = '1'; // Полученный символ

// Проходимся по всем словам

for (int i = 0; i < num\_lines; ++i) {

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

c = getc(file);

if (c == '\n') {

tempory2 = ftell(file);

}

if (!strcmp(line[i], basename) && (tempory2 == ftell(file) || tempory2 == ftell(file) - strlen(line[i]) - 1 || i == 0)) { // Если наша БД, то делаем с ней операции

tempory = 1;

\*pos1 = ftell(file);

\*pos2 = strlen(line[i]);

temp1 = i + 1;

}

if (c == '\n' && tempory == 1) { // Конец нашей БД

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

}

if (feof(file))

break;

}

if (temp1 + 1 == temp2) \*status = 1; // 1 элемент

if (temp1 == temp2 + 1) \*status = 2; // 2 или более элемента

// Добавление в структуру элементов

while (temp1 < temp2) {

char\* value = line[temp1];

char\* key = line[temp1 + 1];

HSET(hashtable, key, value);

temp1 += 2;

}

fclose(file);

for (int i = 0; i < num\_lines; i++) {

free(line[i]);

}

free(line);

return hashtable;

}

==========================================================

table.h

==========================================================

#ifndef TABLE\_H

#define TABLE\_H

typedef struct NodeHashTable {

char\* key;

char\* element;

struct NodeHashTable\* next;

struct NodeHashTable\* prev;

} NodeHashTable;

typedef struct HashTable {

NodeHashTable\*\* nodes;

int count;

} HashTable;

HashTable\* initHashTable();

void HSET(HashTable\* hashtable, char\* key, char\* value);

void HDEL(HashTable\* hashtable, char\* key);

char\* HGET(HashTable\* hashtable, char\* key);

void saveToFileTable(HashTable\* hashtable, const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status);

HashTable\* loadFromFileTable(const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status);

#endif

==========================================================

Python

==========================================================

import string

import random

import socket

from flask import Flask, request, redirect

from flask\_cors import CORS

app = Flask(\_\_name\_\_) # Создание экземпляра приложения Flask

CORS(app) # Разрешение CORS для всего приложения

# Декоратор для обработки GET-запроса по пути '/form'

@app.route('/form', methods=['GET'])

def form():

# Возвращает HTML-форму для ввода URL

return '''

<!DOCTYPE html>

<html>

<body>

<h2>Сокращения URL</h2>

<form id="myForm">

<label for="url">Введите URL:</label><br>

<input type="text" id="url" name="url"><br>

<input type="submit" value="Сократить URL">

</form>

<p id="shortUrl"></p>

<script>

document.getElementById('myForm').addEventListener('submit', function(e) {

e.preventDefault();

var url = document.getElementById('url').value;

if(url.trim() === '') {

document.getElementById('shortUrl').innerText = 'Пожалуйста, введите URL';

return;

}

if(new Blob([url]).size > 50000) {

document.getElementById('shortUrl').innerText = 'URL не должен превышать 50000 байт';

return;

}

fetch('/', {

method: 'POST',

headers: {

'Content-Type': 'text/plain'

},

body: url

}).then(function(response) {

if (!response.ok) {

throw new Error('Ошибка сети при попытке сократить URL');

}

return response.text();

}).then(function(text) {

if (text === 'Error.') {

document.getElementById('shortUrl').innerText = 'Произошла ошибка при обработке вашего запроса';

} else {

document.getElementById('shortUrl').innerText = 'Ваш сокращенный URL: ' + text;

}

}).catch(function(error) {

document.getElementById('shortUrl').innerText = 'Произошла ошибка: ' + error.message;

});

});

</script>

</body>

</html>

'''

# Функция для генерации короткого URL

def generate\_short\_url():

return ''.join(random.choices(string.ascii\_letters + string.digits, k=8))

# Декоратор для обработки POST-запроса по корневому пути '/'

@app.route('/', methods=['POST'])

def shorten\_url():

# Получение данных из запроса и декодирование

url = request.data.decode()

# Создание сокета для подключения к серверу

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

try:

# Попытка подключения к серверу

s.connect((SERVER\_IP, SERVER\_PORT))

except Exception as e:

# В случае ошибки возвращается сообщение об ошибке

return f'Не удалось подключиться к серверу: {str(e)}', 500

# Отправка запроса на получение существующего короткого URL

s.sendall(f'--file urls.data --query HGET short\_urls {url}'.encode())

# Получение ответа от сервера

existing\_url = s.recv(1024).decode()

existing\_url = existing\_url.replace('\n', '').replace('\r', '')

# Обработка ошибки или возврат существующего короткого URL

if existing\_url == "Error.":

s.close()

return 'Error.', 501

if existing\_url != "-> False":

s.close()

return f'{request.url\_root}SU/{existing\_url}'

# Генерация нового короткого URL, если существующего нет

while True:

short\_url = generate\_short\_url()

s.sendall(f'--file urls.data --query HGET urls {short\_url}'.encode())

existing\_url = s.recv(1024).decode()

existing\_url = existing\_url.replace('\n', '').replace('\r', '')

if existing\_url == "Error.":

s.close()

return 'Error.', 502

if existing\_url == "-> False":

s.sendall(f'--file urls.data --query HSET urls {url} {short\_url}'.encode())

if s.recv(1024).decode().replace('\n', '').replace('\r', '') == "Error.":

s.close()

return 'Error.', 503

s.sendall(f'--file urls.data --query HSET short\_urls {short\_url} {url}'.encode())

if s.recv(1024).decode().replace('\n', '').replace('\r', '') == "Error.":

s.close()

return 'Error.', 504

s.close()

return f'{request.url\_root}SU/{short\_url}'

# Декоратор для обработки GET-запроса по пути '/SU/<short\_url>'

@app.route('/SU/<short\_url>', methods=['GET'])

def redirect\_url(short\_url):

# Создание сокета для подключения к серверу

with socket.socket(socket.AF\_INET, socket.SOCK\_STREAM) as s:

try:

# Попытка подключения к серверу

s.connect((SERVER\_IP, SERVER\_PORT))

except Exception as e:

# В случае ошибки возвращается сообщение об ошибке

return f'Не удалось подключиться к серверу: {str(e)}', 505

# Отправка запроса на получение оригинального URL

s.sendall(f'--file urls.data --query HGET urls {short\_url}'.encode())

# Получение ответа от сервера

url = s.recv(1024).decode()

url = url.replace('\n', '').replace('\r', '')

# Обработка ошибки или перенаправление на оригинальный URL

if url == "Error.":

s.close()

return 'Error.', 506

s.close()

return redirect(url, code=302)

if \_\_name\_\_ == '\_\_main\_\_':

SERVER\_IP, SERVER\_PORT = input("Введите IP-адрес и порт сервера (формат: myip:myport): ").split(':')

SERVER\_PORT = int(SERVER\_PORT)

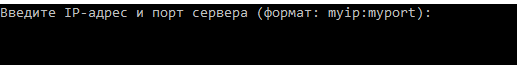
app.run(debug=True)

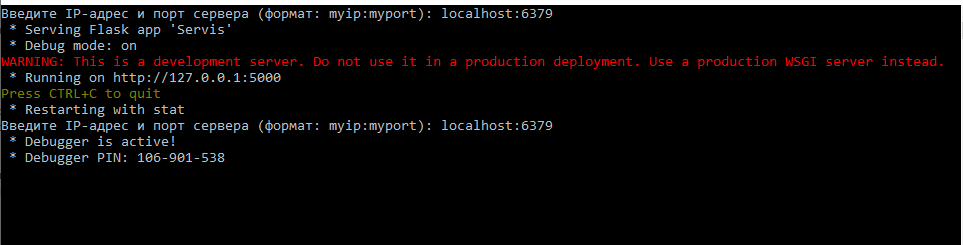
==========================================================**Результат работы программы**

**C**



**Python**





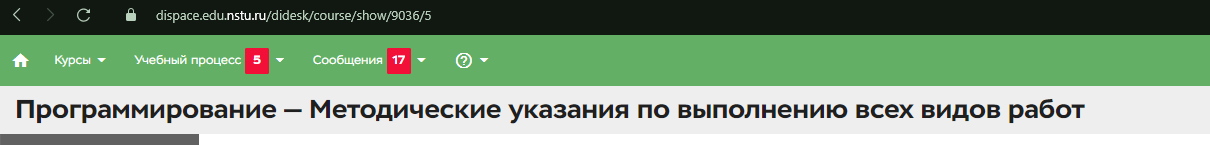
Изображение выглядит как текст, снимок экрана, Шрифт, число

Автоматически созданное описание

Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Автоматически созданное описание





Изображение выглядит как текст, снимок экрана, программное обеспечение, Шрифт

Автоматически созданное описание

**Вывод**: Сервис по сокращению ссылок и сам сервер базы данных были проверены на работоспособность посредством подключения, сначала, к локальному хосту, затем запуском двух программ на разных серверах и железе. Для проверки были вызваны добровольцы, которые пытались положить мои сервера в обычном использовании, однако они устояли и продолжили работу. Все условия задания были учтены. Таким образом, программы являются работоспособными, отвечающими всем требованиям заданию.