МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ государственное БЮДЖЕТНОЕ образовательное учреждениевысшего образования

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»

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

Кафедра защиты информации



**ОТЧЁТ по ПРАКТИЧЕСКОЙ работе №1**

**по дисциплине «ПРОГРАММИРОВАНИЕ»**

**«Разработка системы управления базой»**

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

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

**Цели и задачи работы**: Изучение тем Типы данных, Алгоритмы, Big О нотация.

**Задание к работе**: Разработать no-SQL СУБД, отвечающую следующим требованиям:

1. База данных умеет хранить данные в следующих контейнерах:

1.1 Множество

1.2 Стэк

1.3 Очередь

1.4 Хэш-таблица.

2. Для простоты ЛР необходимо поддерживать только строковый

тип данных.

3. Каждый тип данных поддерживает операцию вставки, доступа

к данным и удаления.

4. Все данные СУБД сохраняет в указанный файл.

5. СУБД имеет консольный интерфейс удовлетворяющий

следующему требованию.

./<имя вашей программы> --file <путь до файла с данными> --

query <запрос к СУБД>

6. Все операции выполняются за O(1)

**Текст программы**

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

С

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

main.c

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

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <locale.h>

#include "stack.h"

#include "set.h"

#include "queue.h"

#include "table.h"

// Функция удаления символов новой строки \n \r

void removeNewline(char\* str);

// Главная функция, принимает аргументы

int main(int argc, char\*\* argv) {

setlocale(LC\_ALL, "Russian");

char\* filename = NULL; // Имя файла

char\* query = NULL;

char\* item = NULL; // Объект

char\* basename = NULL; // Имя БД

char\* key = NULL; // Ключ (Объект)

int temp; // Переменная, отвечающая за номер аргумента

// Проверка количества аргументов

if (argc < 5 || argc > 8 || argv[5] == NULL) {

printf("Неверное количество аргументов.\n");

return 1;

}

// Цикл по аргументам командной строки

for (int i = 1; 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 {

key = argv[i + 3];

item = argv[i + 4];

// Проверка наличия ключа и элемента

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

printf("Ключ или Объект не введены.\n");

return 1;

}

removeNewline(key);

removeNewline(item);

}

if (key != NULL) removeNewline(key);

}

}

// Проверка наличия имени файла и команды

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

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

// Проверка открытия файла

if (!file) {

printf("Такого файла нет!\n");

return 1;

}

removeNewline(basename);

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

if (key == NULL) {

printf("Объект не введён.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

SPUSH(stack, key);

printf("-> %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 (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

char\* element = SPOP(stack);

printf("-> %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) {

printf("Объект не введён.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

SADD(set, key);

printf("-> %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) {

printf("Объект не введён.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

SREM(set, key);

printf("-> %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) {

printf("Объект не введён.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

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

else printf("-> False\n");

fclose(file);

}

}

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

if (key == NULL) {

printf("Объект не введён.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

QPUSH(queue, key);

printf("-> %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 (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

char\* element = QPOP(queue);

printf("-> %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) {

printf("Ключ или Объект не введены.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

HSET(hashtable, key, item);

printf("-> %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) {

printf("Объект не введён.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

HDEL(hashtable, key);

printf("-> %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) {

printf("Объект не введён.\n");

return 1;

}

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

if (pos1 + pos2 == 0) {

printf("Такой базы данных, увы, нет!\n");

fclose(file);

}

else {

if (HGET(hashtable, key) != NULL) printf("%s\n", HGET(hashtable, key));

else printf("-> False\n");

fclose(file);

}

}

}

else {

printf("Не указано имя файла или выполняемая операция.\n");

system("pause");

return 1;

}

system("pause");

return 0;

}

void removeNewline(char\* str) {

if (str != NULL) {

char\* p = str;

while (\*p) {

if (\*p == '\n' || \*p == '\r') {

\*p = "";

}

p++;

}

}

}

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

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; // Счетчик слов

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

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) {

printf("Очередь пуста\n");

return NULL;

}

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");

// Проверка наличия файла

if (file == NULL) {

printf("Ошибка при открытии файла\n");

return;

}

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

// Проверка открытия временного файла

if (tempFile == NULL) {

printf("Ошибка при открытии временного файла\n");

fclose(file);

return;

}

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) {

printf("Ошибка при открытии файла\n");

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 count = 0; // Переменная, отвечающая за количество одинаковых БД

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;

count++;

}

if (feof(file))

break;

}

if (count > 1)

printf("\nТаких баз данных несколько, операции выполняются с последней!\n");

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) {

printf("Элемент уже существует в множестве\n");

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;

}

printf("Элемент не найден в множестве\n");

}

// 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");

// Проверка наличия файла

if (file == NULL) {

printf("Ошибка при открытии файла\n");

return;

}

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

// Проверка открытия временного файла

if (tempFile == NULL) {

printf("Ошибка при открытии временного файла\n");

fclose(file);

return;

}

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) {

printf("Ошибка при открытии файла\n");

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 count = 0; // Переменная, отвечающая за количество одинаковых БД

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;

count++;

}

if (feof(file))

break;

}

if (count > 1)

printf("\nТаких баз данных несколько, операции выполняются с последней!\n");

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) {

printf("Стек пуст\n");

return NULL;

}

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");

// Проверка наличия файла

if (file == NULL) {

printf("Ошибка при открытии файла\n");

return;

}

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

// Проверка открытия временного файла

if (tempFile == NULL) {

printf("Ошибка при открытии временного файла\n");

fclose(file);

return;

}

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) {

printf("Ошибка при открытии файла\n");

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 count = 0; // Переменная, отвечающая за количество одинаковых БД

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;

count++;

}

if (feof(file))

break;

}

if (count > 1)

printf("\nТаких баз данных несколько, операции выполняются с последней!\n");

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);

printf("Ключ уже существует.\n");

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--;

}

else {

printf("Ключ не найден.\n");

}

}

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

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

if (file == NULL) {

printf("Ошибка при открытии файла\n");

return;

}

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

if (tempFile == NULL) {

printf("Ошибка при открытии временного файла\n");

fclose(file);

return;

}

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) {

printf("Ошибка при открытии файла\n");

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 count = 0; // Переменная, отвечающая за количество одинаковых БД

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;

count++;

}

if (feof(file))

break;

}

if (count > 1) printf("\nТаких баз данных несколько, операции выполняются с последней!\n");

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;

if (temp1 == temp2) printf("\nПоследний элемент возможно был удалён, так как он без ключа!\n");

}

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);

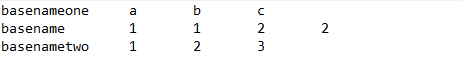
void saveToFileTable(HashTable\* hashtable, const char\* filename, const char\* basename, int\* pos1, int\* pos2, int\* status);

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

#endif

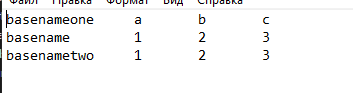
==========================================================**Результат работы программы**

**C**



Изображение выглядит как снимок экрана, текст, Шрифт

Автоматически созданное описание

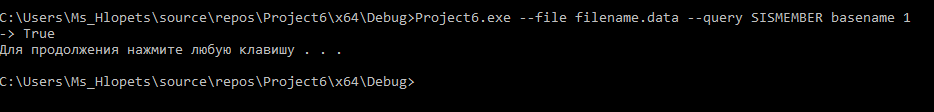


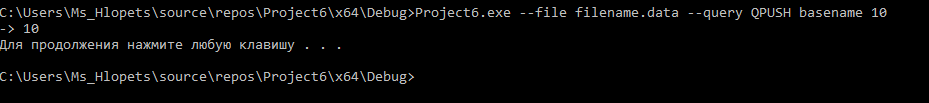
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, линия, Шрифт, снимок экрана

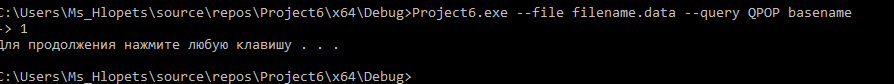
Автоматически созданное описание





Изображение выглядит как текст, снимок экрана, Шрифт, линия

Автоматически созданное описание



Изображение выглядит как текст, снимок экрана, линия, Шрифт

Автоматически созданное описание

**Вывод**: СУБД работает корректно, для этого подтверждения были предоставлены скриншоты работы выше (на них изображены все действия с множеством и очередью).