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 // Отключение предупреждений о небезопасности

#include <stdio.h> // Подключение библиотеки для работы с вводом-выводом

#include <stdlib.h> // Подключение библиотеки для работы с памятью, файлами, случайными числами

#include <string.h> // Подключение библиотеки для работы со строками

#include <locale.h> // Подключение библиотеки для работы с локализацией

#include "stack.h" // Подключение заголовочного файла для работы со стеком

#include "set.h" // Подключение заголовочного файла для работы с множествами

#include "queue.h" // Подключение заголовочного файла для работы с очередью

#include "table.h" // Подключение заголовочного файла для работы с хеш-таблицей

int main(int argc, char\*\* argv) { // Главная функция программы

setlocale(LC\_ALL, "Russian"); // Установка локали на русский язык

char\* filename = NULL; // Имя файла

char\* query = NULL; // Запрос

char\* item = NULL; // Элемент

char\* basename = NULL; // Базовое имя

char\* key = NULL; // Ключ

int temp;

if (argc < 5 || argc > 8) { // Проверка количества аргументов

printf("Неверное количество аргументов.\n"); // Вывод сообщения об ошибке

return 1; // Завершение программы с кодом ошибки

}

for (int i = 1; i < argc; i++) { // Цикл по аргументам командной строки

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

filename = argv[i + 1]; // Присвоение имени файла

}

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

query = argv[i + 1]; // Присвоение запроса

temp = i + 1;

basename = argv[i + 2]; // Присвоение базового имени

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

else {

key = argv[i + 3]; // Присвоение ключа

item = argv[i + 4]; // Присвоение элемента

if (key == NULL || item == NULL) { // Проверка ключа и элемента

printf("Ключ или Объект не введены.\n"); // Вывод сообщения об ошибке

return 1; // Завершение программы с кодом ошибки

}

}

}

}

if (filename != NULL && query != NULL) { // Проверка имени файла и запроса

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

if (!file) { // Проверка файла

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

}

int pos1 = 0;; // Позиция 1

int pos2 = 0;; // Позиция 2

int status = 0;; // Статус

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

if (key == NULL) { // Проверка ключа

printf("Объект не введён.\n"); // Вывод сообщения об ошибке

return 1; // Завершение программы с кодом ошибки

}

Stack\* stack = loadFromFileStack(filename, basename, &pos1, &pos2, &status); // Загрузка стека из файла

if (pos1 + pos2 == 0) { // Проверка позиций

printf("Такой базы данных, увы, нет!\n"); // Вывод сообщения об ошибке

fclose(file); // Закрытие файла

}

else {

SPUSH(stack, key); // Добавление элемента в стек

printf("-> %s\n", key); // Вывод ключа

if (status == 1) status = 0; // Сброс статуса

fclose(file); // Закрытие файла

saveToFileStack(stack, filename, basename, &pos1, &pos2, &status); // Сохранение стека в файл

}

}

// Далее идут аналогичные блоки для других запросов: "SPOP", "SADD", "SREM", "SISMEMBER", "QPUSH", "QPOP", "HSET", "HDEL", "HGET"

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("-> True\n");

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

fclose(file);

}

}

}

else {

printf("Не указано имя файла или выполняемая операция.\n"); // Вывод сообщения об ошибке

system("pause"); // Пауза

return 1; // Завершение программы с кодом ошибки

}

system("pause"); // Пауза

return 0; // Завершение программы с кодом успеха

}

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

queue.c

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

#define \_CRT\_SECURE\_NO\_WARNINGS // Отключение предупреждений о небезопасности

#include "queue.h" // Подключение заголовочного файла для работы с очередью

#include <stdio.h> // Подключение библиотеки для работы с вводом-выводом

#include <stdlib.h> // Подключение библиотеки для работы с памятью, файлами, случайными числами

#include <string.h> // Подключение библиотеки для работы со строками

int countWordsInFileQueue(const char\* filename) { // Функция подсчета слов в файле

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

int count = 0; // Счетчик слов

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

while (fscanf(file, "%s", word) != EOF) { // Цикл по словам в файле

count++; // Увеличение счетчика

}

fclose(file); // Закрытие файла

return count; // Возврат количества слов

}

Queue\* initQueue() { // Функция инициализации очереди

Queue\* queue = (Queue\*)malloc(sizeof(Queue)); // Выделение памяти под очередь

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

}

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

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

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

}

}

char\* QPOP(Queue\* queue) { // Функция удаления элемента из очереди

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

printf("Очередь пуста\n"); // Вывод сообщения об ошибке

return NULL; // Возврат 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

}

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;

fseek(file, 0, SEEK\_SET);

fseek(tempFile, 0, SEEK\_SET);

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

fputc(ch, tempFile);

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

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

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

NodeQueue\* currentNode = queue->front;

while (currentNode != NULL) {

if (currentNode->next == NULL)

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

else

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

currentNode = currentNode->next;

}

if (\*status == 1) {

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

fprintf(tempFile, "\n");

}

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

}

}

free(queue->rear);

free(queue->front);

free(queue);

fclose(file);

fclose(tempFile);

remove(filename);

rename("temp.data", filename);

}

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

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

if (file == NULL) {

return NULL;

}

int num\_lines = countWordsInFileQueue(filename);

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;

int pos3 = 0;

int temp1 = 0;

int temp2 = 0;

char c = '1';

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

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

c = getc(file);

pos3 = ftell(file);

if (!strcmp(line[i], basename)) {

fseek(file, -3 - strlen(line[i]), SEEK\_CUR);

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

fseek(file, pos3, SEEK\_SET);

tempory = 1;

\*pos1 = ftell(file);

temp1 = i + 1;

}

else fseek(file, pos3, SEEK\_SET);

}

if (c == '\n' && tempory == 1) {

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

}

if (feof(file))

break;

}

if (temp1 == temp2)

\*status = 1;

if (temp1 == temp2 + 1)

\*status = 2;

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 // Игнорирование предупреждений о безопасности

#include "set.h" // Подключение заголовочного файла "set.h"

#include <stdio.h> // Подключение стандартной библиотеки ввода-вывода

#include <stdlib.h> // Подключение стандартной библиотеки для работы с памятью, числами и другими

#include <string.h> // Подключение библиотеки для работы со строками

#define MAX\_SIZE 100000 // Определение максимального размера

// Функция для подсчета слов в файле

int countWordsInFileSet(const char\* filename) {

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

int count = 0; // Инициализация счетчика слов

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

while (fscanf(file, "%s", word) != EOF) { // Чтение слов из файла, пока не достигнут конец файла

count++; // Увеличение счетчика на 1

}

fclose(file); // Закрытие файла

return count; // Возврат количества слов

}

// Функция для инициализации множества

Set\* initSet() {

Set\* set = (Set\*)malloc(sizeof(Set)); // Выделение памяти под множество

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\*)); // Выделение памяти под хеш-таблицу

set->emptySlots = (int\*)malloc(MAX\_SIZE \* sizeof(int)); // Выделение памяти под массив пустых слотов

for (int i = 0; i < MAX\_SIZE; i++) { // Цикл по всем элементам хеш-таблицы

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 хеша строки

int calculateHashS(const char\* element) {

int hash = 0; // Инициализация хеша

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

hash = 31 \* hash + element[i]; // Вычисление хеша

}

return abs(hash) % MAX\_SIZE; // Возврат абсолютного значения хеша, ограниченного максимальным размером

}

// Функция для добавления элемента в множество

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

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

if (set->hashTable[hash] != NULL) { // Если элемент уже существует в множестве

printf("Элемент уже существует в множестве\n"); // Вывод сообщения об ошибке

return; // Выход из функции

}

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

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

newNode->hash = hash; // Сохранение хеша в узле

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 перед головой множества

}

set->head = newNode; // Установка нового узла в качестве головы множества

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

set->size++; // Увеличение размера множества на 1

}

// Функция для удаления элемента из множества

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

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

if (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) == 0) { // Если элемент найден в множестве

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; // Выход из функции

}

printf("Элемент не найден в множестве\n"); // Вывод сообщения об ошибке

}

// Функция для проверки наличия элемента в множестве

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

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

if (set->hashTable[hash] != NULL && strcmp(set->hashTable[hash]->element, element) == 0) { // Если элемент найден в множестве

return 1; // Возврат 1

}

return 0; // Возврат 0

}

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;

fseek(file, 0, SEEK\_SET);

fseek(tempFile, 0, SEEK\_SET);

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

Node\* current = set->head;

int i = 0;

while (current != NULL) {

elements[i] = current->element;

current = current->next;

i++;

}

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

fputc(ch, tempFile);

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

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

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

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

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

if (j > 0)

fprintf(tempFile, "\t");

else

fprintf(tempFile, "\n");

}

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("2.data");

rename("temp.data", "2.data");

}

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

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

if (file == NULL) {

return NULL;

}

int num\_lines = countWordsInFileSet(filename);

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;

int pos3 = 0;

int temp1 = 0;

int temp2 = 0;

char c = '1';

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

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

c = getc(file);

pos3 = ftell(file);

if (!strcmp(line[i], basename)) {

fseek(file, -3 - strlen(line[i]), SEEK\_CUR);

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

fseek(file, pos3, SEEK\_SET);

tempory = 1;

\*pos1 = ftell(file);

temp1 = i + 1;

}

else fseek(file, pos3, SEEK\_SET);

}

if (c == '\n' && tempory == 1) {

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

}

if (feof(file))

break;

}

if (temp1 == temp2)

\*status = 1;

if (temp1 == temp2 + 1)

\*status = 2;

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 // Игнорирование предупреждений о безопасности

#include "stack.h" // Подключение заголовочного файла "stack.h"

#include <stdio.h> // Подключение стандартной библиотеки ввода-вывода

#include <stdlib.h> // Подключение стандартной библиотеки для работы с памятью, числами и другими

#include <string.h> // Подключение библиотеки для работы со строками

// Функция для подсчета слов в файле

int countWordsInFileStack(const char\* filename) {

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

int count = 0; // Инициализация счетчика слов

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

while (fscanf(file, "%s", word) != EOF) { // Чтение слов из файла, пока не достигнут конец файла

count++; // Увеличение счетчика на 1

}

fclose(file); // Закрытие файла

return count; // Возврат количества слов

}

// Функция для инициализации стека

Stack\* initStack() {

Stack\* stack = (Stack\*)malloc(sizeof(Stack)); // Выделение памяти под стек

stack->top = NULL; // Инициализация вершины стека

return stack; // Возврат указателя на стек

}

// 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) {

if (stack->top == NULL) { // Если стек пуст

printf("Стек пуст\n"); // Вывод сообщения об ошибке

return NULL; // Возврат 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

}

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;

fseek(file, 0, SEEK\_SET);

fseek(tempFile, 0, SEEK\_SET);

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

fputc(ch, tempFile);

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

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

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

NodeStack\* currentNode = stack->top;

NodeStack\* prevNode = NULL;

while (currentNode != NULL) {

NodeStack\* nextNode = currentNode->next;

currentNode->next = prevNode;

prevNode = currentNode;

currentNode = nextNode;

}

currentNode = prevNode;

while (currentNode != NULL) {

if (currentNode->next == NULL)

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

else

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

currentNode = currentNode->next;

}

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);

}

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

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

if (file == NULL) {

return NULL;

}

int num\_lines = countWordsInFileStack(filename);

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;

int pos3 = 0;

int temp1 = 0;

int temp2 = 0;

char c = '1';

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

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

c = getc(file);

pos3 = ftell(file);

if (!strcmp(line[i], basename)) {

fseek(file, -3 - strlen(line[i]), SEEK\_CUR);

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

fseek(file, pos3, SEEK\_SET);

tempory = 1;

\*pos1 = ftell(file);

temp1 = i + 1;

}

else fseek(file, pos3, SEEK\_SET);

}

if (c == '\n' && tempory == 1) {

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

}

if (feof(file))

break;

}

if (temp1 == temp2)

\*status = 1;

if (temp1 == temp2 + 1)

\*status = 2;

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 // Игнорирование предупреждений о безопасности

#include "table.h" // Подключение заголовочного файла "table.h"

#include <stdio.h> // Подключение стандартной библиотеки ввода-вывода

#include <stdlib.h> // Подключение стандартной библиотеки для работы с памятью, числами и другими

#include <string.h> // Подключение библиотеки для работы со строками

#define MAX\_SIZE 100000 // Определение максимального размера

// Функция для подсчета слов в файле

int countWordsInFileTable(const char\* filename) {

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

int count = 0; // Инициализация счетчика слов

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

while (fscanf(file, "%s", word) != EOF) { // Чтение слов из файла, пока не достигнут конец файла

count++; // Увеличение счетчика на 1

}

fclose(file); // Закрытие файла

return count; // Возврат количества слов

}

// Функция для инициализации хеш-таблицы

HashTable\* initHashTable() {

HashTable\* ht = (HashTable\*)malloc(sizeof(HashTable)); // Выделение памяти под хеш-таблицу

ht->head = NULL; // Инициализация головы хеш-таблицы

ht->size = 0; // Инициализация размера хеш-таблицы

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

ht->hashTable = (NodeHashTable\*\*)malloc(MAX\_SIZE \* sizeof(NodeHashTable\*)); // Выделение памяти под хеш-таблицу

ht->keys = (char\*\*)malloc(MAX\_SIZE \* sizeof(char\*)); // Выделение памяти под ключи

for (int i = 0; i < MAX\_SIZE; i++) { // Цикл по всем элементам хеш-таблицы

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

ht->keys[i] = NULL; // Инициализация ключа

}

return ht; // Возврат указателя на хеш-таблицу

}

// Функция для вычисления хеша строки

int calculateHashT(const char\* element) {

int hash = 0; // Инициализация хеша

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

hash = 31 \* hash + element[i]; // Вычисление хеша

}

return abs(hash) % MAX\_SIZE; // Возврат абсолютного значения хеша, ограниченного максимальным размером

}

// Функция для добавления элемента в хеш-таблицу

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

int hash = calculateHashT(key); // Вычисление хеша ключа

if (ht->hashTable[hash] != NULL) { // Если ключ уже существует в хеш-таблице

printf("Ключ уже существует в хеш-таблице\n"); // Вывод сообщения об ошибке

return; // Выход из функции

}

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

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

newNode->hash = hash; // Сохранение хеша в узле

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

if (ht->head != NULL) { // Если голова хеш-таблицы не пуста

ht->head->prev = newNode; // Установка предыдущего узла перед головой хеш-таблицы

}

ht->head = newNode; // Установка нового узла в качестве головы хеш-таблицы

ht->hashTable[hash] = newNode; // Добавление нового узла в хеш-таблицу

ht->size++; // Увеличение размера хеш-таблицы на 1

ht->keys[ht->size - 1] = \_strdup(key); // Добавление ключа в массив ключей

}

// Функция для получения элемента из хеш-таблицы

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

if (ht->hashTable[calculateHashT(key)] != NULL) { // Если ключ найден в хеш-таблице

return ht->hashTable[calculateHashT(key)]->element; // Возврат элемента

}

return NULL; // Возврат NULL, если ключ не найден

}

// Функция для удаления элемента из хеш-таблицы

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

if (ht->hashTable[calculateHashT(key)] != NULL) { // Если ключ найден в хеш-таблице

NodeHashTable\* nodeToRemove = ht->hashTable[calculateHashT(key)]; // Получение узла для удаления

if (nodeToRemove == ht->head) { // Если узел для удаления является головой хеш-таблицы

ht->head = nodeToRemove->next; // Установка следующего узла в качестве головы хеш-таблицы

}

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

ht->hashTable[calculateHashT(key)] = NULL; // Удаление узла из хеш-таблицы

ht->size--; // Уменьшение размера хеш-таблицы на 1

return; // Выход из функции

}

printf("Ключ не найден в хеш-таблице\n"); // Вывод сообщения об ошибке

}

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;

fseek(file, 0, SEEK\_SET);

fseek(tempFile, 0, SEEK\_SET);

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

fputc(ch, tempFile);

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

fprintf(tempFile, "\t%s\t%s", hashtable->hashTable[calculateHashT(hashtable->keys[0])]->element, hashtable->keys[0]);

}

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

for (int i = 0; i < hashtable->size; i++) {

if (i == hashtable->size - 1) {

fprintf(tempFile, "%s\t%s\n", hashtable->hashTable[calculateHashT(hashtable->keys[i])]->element, hashtable->keys[i]);

}

else {

fprintf(tempFile, "%s\t%s\t", hashtable->hashTable[calculateHashT(hashtable->keys[i])]->element, hashtable->keys[i]);

}

}

if (\*status == 1) {

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

fprintf(tempFile, "\n");

}

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

}

}

free(hashtable->hashTable);

free(hashtable->keys);

free(hashtable);

fclose(file);

fclose(tempFile);

remove(filename);

rename("temp.data", filename);

}

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

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

int num\_lines = countWordsInFileTable(filename);

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;

int tempory2 = 0;

int count = 0;

int temp1 = 0;

int temp2 = 0;

char c = '1';

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)) {

tempory = 1;

\*pos1 = ftell(file);

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

temp1 = i + 1;

}

if (c == '\n' && tempory == 1) {

temp2 = i;

\*pos2 = ftell(file);

tempory = 0;

count++;

}

if (feof(file))

break;

}

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

if (temp1 == temp2 + 1) \*status = 2;

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\* element;

int hash;

struct NodeHashTable\* next;

struct NodeHashTable\* prev;

} NodeHashTable;

typedef struct HashTable {

NodeHashTable\* head;

int size;

NodeHashTable\*\* hashTable;

int tableSize;

char\*\* keys;

} 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);

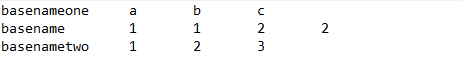
HashTable\* loadFromFileTable(const char\* filename, const char\* basename, int \*pos1, int \*pos2, int \*status);

#endif

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

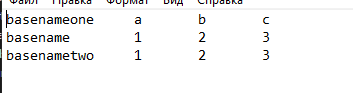
**Результат работы программы**

**C**



Изображение выглядит как снимок экрана, текст, Шрифт

Автоматически созданное описание

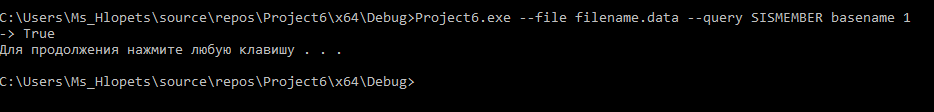


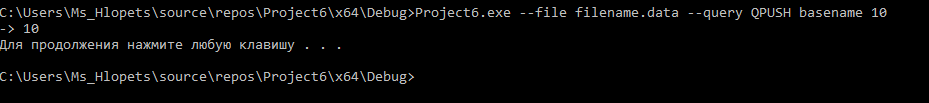
Изображение выглядит как текст, снимок экрана, Шрифт

Автоматически созданное описание

Изображение выглядит как текст, линия, Шрифт, снимок экрана

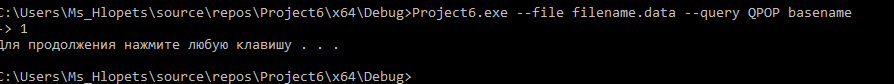
Автоматически созданное описание





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).