Цель работы:

Научиться конструировать пользовательские типы данных — структуры, создавать массивы структур в динамической памяти, осуществлять ввод и ­вывод из файла и консоли.

Описание задачи:

Написать программу, представляющую собой простейшую систему для хранения и управления данными. Программа должна поддерживать операции вставки, удаления, обновления и поиска данных. Данные должны представлять собой линейный список. Команды для выполнения должны подаваться как из файла так и из командной строки. Результаты должны выводится как в консоль так и в файл. Описание команд и формата их задания определяются вариантом задания. В соответствии со своим вариантом следует сконструировать схему базы данных.

Подходы к решению:

Разбиение программы на импортируемые модули, что позволило удобнее отлаживать код и работать с меньшим его объемом;

Структуры данных и алгоритмы:

Типы используемых данных:

* Char;
* Int;
* Двумерные массивы;
* Одномерные массивы;

Линейный односвязный список

* Алгоритм вставки, основанный на проверке наличия необходимых атрибутов и добавления элемента в конец списка;
* Алгоритм поиска, основанный на переборе всех элементов списка при соответствии заданным условиям;
* Алгоритм удаления, основанный на проверке элементов заданным условием и высвобождением памяти;
* Алгоритм удаления дублирующихся строк, основанный на поэлементном поиске дубля для каждой строки и его удалением в случае совпадения.

Пользовательский тип данных(вариант 8) - структура представляющая строку БД:

struct row {

    char last\_name[20];

    char first\_name[20];

    char middle\_name[20];

    char number[15];

    char services[MAX\_SERVICES][SERVICE\_NAME\_LENGTH];

    int discount\_id;

    int bonus\_id;

};

last\_name – фамилия пользователя;

first\_name – имя пользователя;

middle\_name – отчество пользователя;

number – номер телефона пользователя;

discount\_id – идентификаторы скидок;

bonus\_id – идентификаторы бонусов;

services – список подключенных услуг;

Выводы:

В ходе выполнения задания были получены практические навыки работы с односвязными списками, указателями, динамическим выделением памяти, чтением и записью в файл, чтением и выводом в консоль.

Текст программы:

delete.h:

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "mem\_calc.h"

#include "list.h"

#include "filter.h"

struct node\* exec\_delete(struct node\* head, const char\* fltr) {

    int first = 1;

    int cnt=0;

    int i = 0;

    struct node\* head\_base = head;

    while (head != NULL) {

        if (filter(fltr, head->data) == 1) {

            delete\_node(&head\_base, i);

            head = head\_base;

            cnt++;

            continue;

        }

        head = head->next;

        i++;

    }

    printf("deleted: %d rows\n", cnt);

    return head\_base;

}

filter.h:

#pragma once

#include <stdlib.h>

#include <string.h>

#include <stdio.h>

#include <ctype.h>

#include "mem\_calc.h"

#include "list.h"

#define FILTER\_ARRAY\_SIZE 100

void get\_field(const char\* cond, char field[15])

{

    for (int i=0; i<15; i++)

        field[i] = '\0';

    for(int i = 0; i < strlen(cond); i++)

    {

        char t = cond[i];

        if(isalpha(t) == 0 && t != '\_')

        {

            strncpy(field, cond, i);

            break;

        }

    }

}

void get\_value(const char\* cond, int trunc\_length, char value[50])

{

    for (int i=0; i<50; i++)

        value[i] = '\0';

    strncpy(value, cond + trunc\_length, strlen(cond) - trunc\_length + 1);

}

void get\_value\_array(const char\* cond, int trunc\_length, char res\_arr[FILTER\_ARRAY\_SIZE][12])

{

    for (int i=0; i<FILTER\_ARRAY\_SIZE; i++)

        for (int j=0; j<12; j++)

            res\_arr[i][j] = '\0';

    int start = 0;

    int j = 0;

    for(int i = trunc\_length; i < strlen(cond); i++)

    {

        char t = cond[i];

        if(t == '\'' && start == 0)

        {

            start = i;

        }

        else if(t == '\'' && start != 0)

        {

            strncpy(res\_arr[j], &(cond[start+1]), i - start - 1);

            start = 0;

            j++;

        }

    }

}

int str\_comparison(const char\* comparison\_operand, const char\* val1, const char\* val2)

{

    if (strcmp(comparison\_operand, ">") == 0)

    {

        if(strcmp(val1, val2) > 0)

        {

            return 1;

        }

        return 0;

    }

    if (strcmp(comparison\_operand, "<") == 0)

    {

        if(strcmp(val1, val2) < 0)

        {

            return 1;

        }

        return 0;

    }

    if (strcmp(comparison\_operand, "==") == 0)

    {

        if(strcmp(val1, val2) == 0)

        {

            return 1;

        }

        return 0;

    }

    if (strcmp(comparison\_operand, "!=") == 0)

    {

        if(strcmp(val1, val2) != 0)

        {

            return 1;

        }

        return 0;

    }

    return 0;

}

int ulong\_comparison(const char\* comparison\_operand, unsigned long val1, unsigned long val2)

{

    if (strcmp(comparison\_operand, ">") == 0)

    {

        return val1 > val2;

    }

    if (strcmp(comparison\_operand, "<") == 0)

    {

        return val1 < val2;

    }

    if (strcmp(comparison\_operand, "==") == 0)

    {

        return val1 == val2;

    }

    if (strcmp(comparison\_operand, "!=") == 0)

    {

        return val1 != val2;

    }

    return 0;

}

int except(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char arr[FILTER\_ARRAY\_SIZE][12];

    get\_value\_array(cond, strlen(field) + 3, arr);

    int hit = 0;

    int count = 0;

    for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

    {

        if(strcmp(arr[i], "\0") != 0)

        {

            count += 1;

        }

        else

        {

            break;

        }

    }

    if(strcmp(field, "services") == 0)

    {

        for(int j = 0; j < MAX\_SERVICES; j++)

        {

            for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

            {

                if(strcmp(arr[i], r.services[j]) == 0 && strcmp(arr[i], "\0") != 0)

                {

                    hit += 1;

                    break;

                }

                if(strcmp(arr[i], "\0") == 0)

                {

                    break;

                }

            }

        }

        return count != hit;

    }

    return 0;

}

int only\_include(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char arr[FILTER\_ARRAY\_SIZE][12];

    get\_value\_array(cond, strlen(field) + 3, arr);

    int hit = 0;

    int count = 0;

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

    {

        if(strcmp(r.services[i], "\0") != 0)

        {

            count += 1;

        }

        else

        {

            break;

        }

    }

    if(strcmp(field, "services") == 0)

    {

        for(int j = 0; j < MAX\_SERVICES; j++)

        {

            for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

            {

                if(strcmp(arr[i], r.services[j]) == 0 && strcmp(arr[i], "\0") != 0)

                {

                    hit += 1;

                    break;

                }

                if(strcmp(arr[i], "\0") == 0)

                {

                    break;

                }

            }

        }

        return count == hit;

    }

    return 0;

}

int equal(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char value[50];

    get\_value(cond, strlen(field) + 2, value);

    if(strcmp(field, "first\_name") == 0)

    {

        return str\_comparison("==", r.first\_name, value);

    }

    if(strcmp(field, "last\_name") == 0)

    {

        return str\_comparison("==", r.last\_name, value);

    }

    if(strcmp(field, "middle\_name") == 0)

    {

        return str\_comparison("==", r.middle\_name, value);

    }

    if(strcmp(field, "number") == 0)

    {

        unsigned long converted\_val1;

        unsigned long converted\_val2;

        sscanf(r.number, "%d", &converted\_val1);

        sscanf(value, "%d", &converted\_val2);

        return ulong\_comparison("==", converted\_val1, converted\_val2);

    }

    if(strcmp(field, "discount\_id") == 0)

    {

        return r.discount\_id == atoi(value);

    }

    if(strcmp(field, "bonus\_id") == 0)

    {

        return r.bonus\_id == atoi(value);

    }

    return 0;

}

int not\_equal(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char value[50];

    get\_value(cond, strlen(field) + 2, value);

    if(strcmp(field, "first\_name") == 0)

    {

        return str\_comparison("!=", r.first\_name, value);

    }

    if(strcmp(field, "last\_name") == 0)

    {

        return str\_comparison("!=", r.last\_name, value);

    }

    if(strcmp(field, "middle\_name") == 0)

    {

        return str\_comparison("!=", r.middle\_name, value);

    }

    if(strcmp(field, "number") == 0)

    {

        unsigned long converted\_val1;

        unsigned long converted\_val2;

        sscanf(r.number, "%d", &converted\_val1);

        sscanf(r.number, "%d", &converted\_val2);

        return ulong\_comparison("!=", converted\_val1, converted\_val2);

    }

    if(strcmp(field, "discount\_id") == 0)

    {

        return r.discount\_id != atoi(value);

    }

    if(strcmp(field, "bonus\_id") == 0)

    {

        return r.bonus\_id != atoi(value);

    }

    return 0;

}

int more(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char value[50];

    get\_value(cond, strlen(field) + 1, value);

    if(strcmp(field, "first\_name") == 0)

    {

        return str\_comparison(">", r.first\_name, value);

    }

    if(strcmp(field, "last\_name") == 0)

    {

        return str\_comparison(">", r.last\_name, value);

    }

    if(strcmp(field, "middle\_name") == 0)

    {

        return str\_comparison(">", r.middle\_name, value);

    }

    if(strcmp(field, "number") == 0)

    {

        unsigned long converted\_val1;

        unsigned long converted\_val2;

        sscanf(r.number, "%d", &converted\_val1);

        sscanf(r.number, "%d", &converted\_val2);

        return ulong\_comparison(">", converted\_val1, converted\_val2);

    }

    if(strcmp(field, "discount\_id") == 0)

    {

        return r.discount\_id > atoi(value);

    }

    if(strcmp(field, "bonus\_id") == 0)

    {

        return r.bonus\_id > atoi(value);

    }

    return 0;

}

int less(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char value[50];

    get\_value(cond, strlen(field) + 1, value);

    if(strcmp(field, "first\_name") == 0)

    {

        return str\_comparison("<", r.first\_name, value);

    }

    if(strcmp(field, "last\_name") == 0)

    {

        return str\_comparison("<", r.last\_name, value);

    }

    if(strcmp(field, "middle\_name") == 0)

    {

        return str\_comparison("<", r.middle\_name, value);

    }

    if(strcmp(field, "number") == 0)

    {

        unsigned long converted\_val1;

        unsigned long converted\_val2;

        sscanf(r.number, "%d", &converted\_val1);

        sscanf(r.number, "%d", &converted\_val2);

        return ulong\_comparison("<", converted\_val1, converted\_val2);

    }

    if(strcmp(field, "discount\_id") == 0)

    {

        return r.discount\_id < atoi(value);

    }

    if(strcmp(field, "bonus\_id") == 0)

    {

        return r.bonus\_id < atoi(value);

    }

    return 0;

}

int include(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char arr[FILTER\_ARRAY\_SIZE][12];

    get\_value\_array(cond, strlen(field) + 3, arr);

    int hit = 0;

    int count = 0;

    for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

    {

        if(strcmp(arr[i], "\0") != 0)

        {

            count += 1;

        }

        else

        {

            break;

        }

    }

    if(strcmp(field, "services") == 0)

    {

        for(int j = 0; j < MAX\_SERVICES; j++)

        {

            for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

            {

                if(strcmp(arr[i], r.services[j]) == 0  && strcmp(arr[i], "\0") != 0)

                {

                    hit += 1;

                    break;

                }

                if(strcmp(arr[i], "\0") == 0)

                {

                    break;

                }

            }

        }

        return count == hit;

    }

    return 0;

}

int in(const char\* cond, struct row r)

{

    char field[15];

    get\_field(cond, field);

    char arr[FILTER\_ARRAY\_SIZE][12];

    get\_value\_array(cond, strlen(field) + 3, arr);

    if(strcmp(field, "first\_name") == 0)

    {

        for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

        {

            if(strcmp(arr[i], r.first\_name) == 0  && strcmp(arr[i], "\0") != 0)

            {

                return 1;

            }

            if(strcmp(arr[i], "\0") == 0)

            {

                break;

            }

        }

        return 0;

    }

    if(strcmp(field, "last\_name") == 0)

    {

        for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

        {

            if(strcmp(arr[i], r.last\_name) == 0  && strcmp(arr[i], "\0") != 0)

            {

                return 1;

            }

            if(strcmp(arr[i], "\0") == 0)

            {

                break;

            }

        }

        return 0;

    }

    if(strcmp(field, "middle\_name") == 0)

    {

        for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

        {

            if(strcmp(arr[i], r.middle\_name) == 0  && strcmp(arr[i], "\0") != 0)

            {

                return 1;

            }

            if(strcmp(arr[i], "\0") == 0)

            {

                break;

            }

        }

        return 0;

    }

    if(strcmp(field, "services") == 0)

    {

        for(int j = 0; j < MAX\_SERVICES; j++)

        {

            for(int i = 0; i < FILTER\_ARRAY\_SIZE; i++)

            {

                if(strcmp(arr[i], r.services[j]) == 0  && strcmp(arr[i], "\0") != 0)

                {

                    return 1;

                }

                if(strcmp(arr[i], "\0") == 0)

                {

                    break;

                }

            }

        }

        return 0;

    }

    return 0;

}

int do\_work(const char\* cond, struct row r)

{

    if(strstr(cond, ">"))

    {

        return more(cond, r);

    }

    else if(strstr(cond, "<"))

    {

        return less(cond, r);

    }

    else if(strstr(cond, "=="))

    {

        if(strstr(cond, "["))

        {

            return only\_include(cond, r);

        }

        else

        {

            return equal(cond, r);

        }

    }

    else if(strstr(cond, "!="))

    {

        if(strstr(cond, "["))

        {

            return except(cond, r);

        }

        else

        {

            return not\_equal(cond, r);

        }

    }

    else if(strstr(cond, "/in/"))

    {

        return in(cond, r);

    }

    else if(strstr(cond, "/include/"))

    {

        return include(cond, r);

    }

    else

    {

        return 0;

    }

}

int filter(const char\* cond, struct row r)

{

    int result;

    int len\_cond = strlen(cond);

    if (len\_cond == 0)

    {

        return 1;

    }

    else

    {

        const char\* del\_serv = " ";

        char tmp[1024];

        strcpy(tmp, cond);

        char\* service = strtok(tmp, del\_serv);

        char single\_cond[20][50];

        int i = 0;

        while (service) {

            strcpy(single\_cond[i], service);

            service = strtok(NULL, del\_serv);

            i++;

        }

        for (int j=0; j<i; j++)

        {

            result = do\_work(single\_cond[j], r);

            if(result == 0)

            {

                return 0;

            }

        }

    }

    return 1;

}

insert.h:

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "mem\_calc.h"

#include "list.h"

struct row\* row\_to\_insert(const char\* fields){

    struct row\* row = (struct row\*)malloc(sizeof(struct row));

    const char\* del\_serv = ",";

    char tmp[512];

    strcpy(tmp, fields);

    char\* service = strtok(tmp, del\_serv);

    char single\_fld[20][30];

    int i = 0;

    while (service) {

        strcpy(single\_fld[i], service);

        service = strtok(NULL, del\_serv);

        i++;

    }

    int k1 = 0;

    int k2 = 0;

    int k3 = 0;

    int k4 = 0;

    int k5 = 0;

    int k6 = 0;

    for (int j = 0; j < i; j++) {

        const char\* del\_serv = "=";

        char\* field = strtok(single\_fld[j], del\_serv);

        char\* value = strtok(NULL, del\_serv);

        if (strcmp(field, "last\_name") == 0) {

            strcpy(row->last\_name, value);

            k1++;

        }

        else if (strcmp(field, "first\_name") == 0) {

            strcpy(row->first\_name, value);

            k2++;

        }

        else if (strcmp(field, "middle\_name") == 0) {

            strcpy(row->middle\_name, value);

            k3++;

        }

        else if (strcmp(field, "number") == 0) {

            strcpy(row->number, value);

            k4++;

        }

        else if (strcmp(field, "bonus\_id") == 0) {

            row->bonus\_id = atoi(value);

            k5++;

        }

        else if (strcmp(field, "discount\_id") == 0) {

            row->discount\_id = atoi(value);

            k6++;

        }

        else {

            printf("Insert error, unknown field\n");

            return NULL;

        }

    }

    for (int k=0; k < MAX\_SERVICES; ++k)

        strncpy(row->services[k], "\0", 1);

    if ((k1==1) && (k2==1) && (k3==1) && (k4==1) && (k5==1) && (k6==1)){

        return row;}

    else return NULL;

}

void exec\_insert(struct node\* head, const char\* fields) {

    struct row\* r = row\_to\_insert(fields);

    if (r != NULL)

        insert\_back(&head, \*r);

        free(r);

}

list.h:

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "mem\_calc.h"

#define MAX\_SERVICES 14

#define SERVICE\_NAME\_LENGTH 8

struct row {

    char last\_name[20];

    char first\_name[20];

    char middle\_name[20];

    char number[15];

    char services[MAX\_SERVICES][SERVICE\_NAME\_LENGTH];

    int discount\_id;

    int bonus\_id;

};

struct node {

    struct row data;

    struct node\* next;

};

void insert\_front(struct node\*\* head, struct row data) {

    struct node\* new\_node = (struct node\*)malloc(sizeof(struct node));

    if (new\_node == NULL) {

        printf("Error allocating memory\n");

        exit(1);

    }

    new\_node->data = data;

    new\_node->next = \*head;

    \*head = new\_node;

}

void insert\_back(struct node\*\* head, struct row data) {

    struct node\* new\_node = (struct node\*)malloc(sizeof(struct node));

    if (new\_node == NULL) {

        printf("Error allocating memory\n");

        exit(1);

    }

    new\_node->data = data;

    new\_node->next = NULL;

    if (\*head == NULL) {

        \*head = new\_node;

    } else {

        struct node\* curr = \*head;

        while (curr->next != NULL) {

            curr = curr->next;

        }

        curr->next = new\_node;

    }

}

struct node\* get\_node(struct node\* head, int n) {

    int counter = 0;

    while (counter < n && head) {

        head = head->next;

        counter++;

    }

    return head;

}

int get\_list\_size(struct node\* head) {

    int counter = 0;

    struct node\* hd = head;

    while (hd) {

        hd = hd->next;

        counter++;

    }

    return counter;

}

void print\_node(struct node\* head, int n) {

    int counter = 0;

    while (counter < n && head) {

        head = head->next;

        counter++;

    }

    printf("%s %s %s, номер %s, дисконт: %d, бонус: %d, услуги:", head->data.last\_name, head->data.first\_name,

            head->data.middle\_name, head->data.number, head->data.discount\_id, head->data.bonus\_id);

        for (int j = 0; j < MAX\_SERVICES; j++) {

            if (head->data.services[j][0] != '\0') {

                printf(" %s", head->data.services[j]);

            } else {

                break;

            }

        }

        printf("\n");

}

struct row pop(struct node \*\*head) {

    struct node\* prev = NULL;

    struct row val;

    if (head == NULL) {

        exit(-1);

    }

    prev = (\*head);

    val = prev->data;

    (\*head) = (\*head)->next;

    free(prev);

    return val;

}

struct row delete\_node(struct node\*\* head, int n) {

    if (n == 0) {

        struct node\* prev = get\_node(\*head, 0);

        (\*head) = (\*head)->next;

        free(prev);

    } else {

        struct node \*prev = get\_node(\*head, n-1);

        struct node \*elm  = prev->next;

        struct row val = elm->data;

        prev->next = elm->next;

        free(elm);

        return val;

    }

}

void print\_list(struct node\* head) {

    while (head != NULL) {

        printf("last: %s first: %s mid: %s, number %s, discount: %d, bonus: %d, services:", head->data.last\_name, head->data.first\_name,

            head->data.middle\_name, head->data.number, head->data.discount\_id, head->data.bonus\_id);

        for (int j = 0; j < MAX\_SERVICES; j++) {

            if (head->data.services[j][0] != '\0') {

                printf(" %s", head->data.services[j]);

            } else {

                break;

            }

        }

        printf("\n");

        head = head->next;

    }

}

void free\_list(struct node\* head) {

    while (head != NULL) {

        struct node\* temp = head;

        head = head->next;

        free(temp);

    }

}

Mem\_calc.h:

#pragma once

#include <stdlib.h>

#include <stdio.h>

int c\_malloc = 0;

int c\_free = 0;

int c\_calloc = 0;

int c\_realloc = 0;

void\* \_malloc(size\_t size) {

    c\_malloc++;

    return malloc(size);

}

void \_free(void\* ptr) {

    c\_free++;

    return free(ptr);

}

void\* \_calloc(size\_t a, size\_t b) {

    c\_calloc++;

    return calloc(a, b);

}

void\* \_realloc(void\* ptr, size\_t size) {

    if (ptr == NULL)

        c\_malloc++;

    else

        c\_realloc++;

    return realloc(ptr, size);

}

#define malloc(size) \

    \_malloc(size)

#define free(size) \

    \_free(size)

#define calloc(a, b) \

    \_calloc(a, b)

#define realloc(ptr, size) \

    \_realloc(ptr, size)

void print\_mem() {

    printf("\n");

    printf("malloc:%d\n", c\_malloc);

    printf("realloc:%d\n", c\_realloc);

    printf("calloc:%d\n", c\_calloc);

    printf("free:%d\n", c\_free);

    FILE\* file = fopen("memstat.txt", "w");

    fprintf(file, "malloc:%d\n", c\_malloc);

    fprintf(file, "realloc:%d\n", c\_realloc);

    fprintf(file, "calloc:%d\n", c\_calloc);

    fprintf(file, "free:%d\n", c\_free);

    fclose(file);

    return;

}

Parse\_command.h:

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "mem\_calc.h"

struct command{

    int command;

    char fields[100];

    char cond[100];

};

// Функция: замена подстроки в строке на заданную последовательность символов - C (СИ)

char\* find\_and\_insert(char\* str,char\* sub\_str, char\* sub\_str\_rep){

  const int BUFFER\_SIZE = 1024;

  char\* buffer = NULL;

  int buffer\_len = 0;

  // если входных данных нет - ничего не делаем.

  if (str && sub\_str && sub\_str\_rep){

    // предпологаем что под строку str отведенно достаточно памяти.

    // в противном случае ничего не делаем

    // если длина строки str достаточна для хранения новой строки, продолжаем

    if( strlen(str) < strlen(str) - strlen(sub\_str) + strlen(sub\_str\_rep) ){

      // временный массив

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

      // чистим массив от мусора

      memset(buffer, 0, strlen(str));

      // Делаем дело. strlen strstr strcat strcpy и strncpy

      // стандартные ф-ии в string.h

      strncpy(buffer, str, strlen(str) - strlen(strstr(str,sub\_str)) );

      strcat(buffer, sub\_str\_rep);

      strcat(buffer, strstr(str,sub\_str) + strlen(sub\_str));

      strcpy(str, buffer);

      // освобождаем память отведённую под массив

      free(buffer);

    }

  }

  return str;

}

struct command parse\_command(const char\* str1) {

    struct command comm = {-1, "\0", 0, "\0", 0};

    char str\_t[1024];

    strcpy(str\_t, str1);

    char \* token = strtok(str\_t, " ");

    if (strcmp(token, "insert") == 0){

        comm.command = 1;

        char \* tmp = strtok(NULL, " ");

        if (tmp)

            strcpy(comm.fields, tmp);

        else printf("incorrect command: %s\n", str1);

    }

    else if (strcmp(token, "select") == 0){

        comm.command = 2;

        char \* tmp = strtok(NULL, " ");

        if (tmp)

            strcpy(comm.fields, tmp);

            tmp = strtok(NULL, "");

            if (tmp)

                strcpy(comm.cond, tmp);

            else printf("incorrect command: %s\n", str1);

    }

    else if (strcmp(token, "delete") == 0) {

        comm.command = 3;

        char \* tmp = strtok(NULL, "");

        if (tmp)

            strcpy(comm.cond, tmp);

        else printf("incorrect command: %s\n", str1);

    }

    else if (strcmp(token, "update") == 0) {

        comm.command = 4;

        char \* tmp = strtok(NULL, " ");

        if (tmp)

            strcpy(comm.fields, tmp);

            tmp = strtok(NULL, "");

            if (tmp)

                strcpy(comm.cond, tmp);

            else printf("incorrect command: %s\n", str1);

    }

    else if (strcmp(token, "uniq") == 0) {

        comm.command = 5;

        char \* tmp = strtok(NULL, " ");

        if (tmp)

            strcpy(comm.fields, tmp);

        else printf("incorrect command: %s\n", str1);

    }

    //проверка на атрибуты

    switch (comm.command){

        case 1: if (comm.fields == NULL) {

                    comm.command = -1;

                }

                break;

        case 2: if ((comm.fields == NULL)||(comm.cond == NULL)){

                    comm.command = -1;

                }

                break;

        case 3: if (comm.cond == NULL){

                    comm.command = -1;

                }

                break;

        case 4: if ((comm.fields == NULL)||(comm.cond == NULL)) {

                    comm.command = -1;

                }

                break;

        case 5: if (comm.fields == NULL) {

                    comm.command = -1;

                }

                break;

    }

    if (comm.command == 1 || comm.command == 2 || comm.command == 4 || comm.command == 5) {

        int len = strlen(comm.fields);

        if(len > 0)

            if(comm.fields[len-1] == '\n')

                comm.fields[len-1] = 0;

    }

    if (comm.command == 2 || comm.command == 3 || comm.command == 4) {

        int len = strlen(comm.cond);

        if(len > 0)

            if(comm.cond[len-1] == '\n')

                comm.cond[len-1] = 0;

    }

    return comm;

}

Read.h:

#include <stdlib.h>

#include "mem\_calc.h"

#include "parse\_command.h"

#include "list.h"

#include "select.h"

#include "filter.h"

#include "delete.h"

#include "update.h"

#include "uniq.h"

#include "insert.h"

// чтение и выполнение команд

struct node\* read\_command(struct node\* head) {

    char buffer[256];

    FILE \* f = fopen("input.txt","r");

    if(f) {

        while((fgets(buffer, 256, f)) != NULL) {

            struct command a = parse\_command(buffer);

                if (a.command != -1) {

                    //INSERT

                    if (a.command == 1) {

                        exec\_insert(head, a.fields);

                        int cnt = get\_list\_size(head);

                        printf("total: %d\n", cnt);

                    }

                    // SELECT

                    if (a.command == 2) {

                        exec\_select(head, a.cond, a.fields);

                    }

                    // DELETE

                    if (a.command == 3) {

                        head = exec\_delete(head, a.cond);

                    }

                    // UPDATE

                    if (a.command == 4) {

                        int cnt = exec\_update(head, a.cond, a.fields);

                        printf("update: %d\n", cnt);

                    }

                    // UNIQ

                    if (a.command == 5) {

                        exec\_uniq(head, a.fields);

                    }

                }

        }

        fclose(f);

    }

    return head;

};

Select.h:

#pragma once

#include "mem\_calc.h"

#include "list.h"

#include "filter.h"

void exec\_select(struct node\* head, const char\* cond, const char\* fields) {

    struct node\* hd = head;

    int cnt = 0;

    while (hd != NULL) {

        if (filter(cond, hd->data) == 1){

            cnt++;

            const char\* del\_serv = ",";

            char tmp[1024];

            strcpy(tmp, fields);

            char\* service = strtok(tmp, del\_serv);

            char flds[10][30];

            int i = 0;

            while (service) {

                strcpy(flds[i], service);

                service = strtok(NULL, del\_serv);

                i++;

            }

            for (int j=0; j<i; j++) {

                if (strcmp(flds[j], "last\_name") == 0)

                    printf("%s=%s ", flds[j], hd->data.last\_name);

                if (strcmp(flds[j], "first\_name") == 0)

                    printf("%s=%s ", flds[j], hd->data.first\_name);

                if (strcmp(flds[j], "middle\_name") == 0)

                    printf("%s=%s ", flds[j], hd->data.middle\_name);

                if (strcmp(flds[j], "number") == 0)

                    printf("%s=%s ", flds[j], hd->data.number);

                if (strcmp(flds[j], "bonus\_id") == 0)

                    printf("%s=%d ", flds[j], hd->data.bonus\_id);

                if (strcmp(flds[j], "discount\_id") == 0)

                    printf("%s=%d ", flds[j], hd->data.discount\_id);

                if (strcmp(flds[j], "services") == 0) {

                    printf("%s=[", flds[j]);

                    int first=1;

                    for (int k = 0; k < MAX\_SERVICES; k++) {

                        if (hd->data.services[k][0] != '\0') {

                            if (first == 1) {

                                printf("%s", hd->data.services[k]);

                                first = 0;

                            }

                            else

                                printf(",%s", hd->data.services[k]);

                        } else {

                            break;

                        }

                    }

                    printf("] ");

                }

            }

            printf("\n");

        }

        hd = hd->next;

    }

    printf("select: %d\n", cnt);

}

Uniq.h:

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "mem\_calc.h"

#include "list.h"

#include "filter.h"

#define FILTER\_ARRAY\_SIZE 100

int uniq(struct node\* head, const char\* fields, int n)

{

    if(strlen(fields) == 0) return 0;

    struct node\* hd = head;

    const char\* del\_serv = ",";

    char tmp[1024];

    strcpy(tmp, fields);

    char\* service = strtok(tmp, del\_serv);

    char flds[10][30];

    char filter\_cond[1024];

    for (int i=0; i<1024; i++)

        filter\_cond[i] = '\0';

    int i = 0;

    while (service) {

        strcpy(flds[i], service);

        service = strtok(NULL, del\_serv);

        i++;

    }

    for (int j=0; j<i; j++) {

        if (strcmp(flds[j], "last\_name") == 0)

        {

            strcat(filter\_cond, "last\_name==");

            strcat(filter\_cond, hd->data.last\_name);

            strcat(filter\_cond, " ");

        }

        if (strcmp(flds[j], "first\_name") == 0)

        {

            strcat(filter\_cond, "first\_name==");

            strcat(filter\_cond, hd->data.first\_name);

            strcat(filter\_cond, " ");

        }

        if (strcmp(flds[j], "middle\_name") == 0)

        {

            strcat(filter\_cond, "middle\_name==");

            strcat(filter\_cond, hd->data.middle\_name);

            strcat(filter\_cond, " ");

        }

        if (strcmp(flds[j], "number") == 0)

        {

            strcat(filter\_cond, "number==");

            strcat(filter\_cond, hd->data.number);

            strcat(filter\_cond, " ");

        }

        if (strcmp(flds[j], "bonus\_id") == 0)

        {

            char bonus\_id[12];

            itoa(hd->data.bonus\_id, bonus\_id, 10);

            strcat(filter\_cond, "bonus\_id==");

            strcat(filter\_cond, bonus\_id);

            strcat(filter\_cond, " ");

        }

        if (strcmp(flds[j], "discount\_id") == 0)

        {

            char discount\_id[12];

            itoa(hd->data.discount\_id, discount\_id, 10);

            strcat(filter\_cond, "discount\_id==");

            strcat(filter\_cond, discount\_id);

            strcat(filter\_cond, " ");

        }

        if (strcmp(flds[j], "services") == 0)

        {

            strcat(filter\_cond, "services==[");

            for (int k = 0; k <= MAX\_SERVICES; k++)

            {

                if (hd->data.services[k][0] != '\0') {

                    strcat(filter\_cond, "'");

                    strcat(filter\_cond, hd->data.services[k]);

                    strcat(filter\_cond, "'");

                }

                else

                {

                    strcat(filter\_cond, "]");

                    break;

                }

                if(k != MAX\_SERVICES && hd->data.services[k + 1][0] != '\0')

                {

                    strcat(filter\_cond, ",");

                }

            }

        }

    }

    struct node\* nested\_head = hd->next;

    while (nested\_head != NULL) {

        if(filter(filter\_cond, nested\_head->data))

        {

            return 1;

        }

        nested\_head = nested\_head->next;

    }

    return 0;

}

int exec\_uniq(struct node\* head, const char\* fields) {

    int first = 1;

    int cnt=0;

    int i = 0;

    struct node\* hd = head;

    struct node\* head\_base = hd;

    while (hd != NULL) {

        if (uniq(hd, fields, i) == 1) {

            delete\_node(&head\_base, i);

            head = head\_base;

            hd = head;

            cnt++;

            i--;

        }

        hd = hd->next;

        i++;

    }

    printf("uniq: %d\n", cnt);

    return cnt;

}

Update.h:

#pragma once

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "mem\_calc.h"

#include "list.h"

#include "filter.h"

int exec\_update(struct node\* head, const char\* fltr, const char\* fld) {

    int i = 0;

    int res = 0;

    struct node\* hd = head;

    while (hd != NULL) {

        if (filter(fltr, hd->data) == 1){

            res++;

            const char\* del\_serv = ",";

            char tmp[1024];

            strcpy(tmp, fld);

            char\* service = strtok(tmp, del\_serv);

            char single\_fld[20][30];

            int num = 0;

            while (service) {

                strcpy(single\_fld[num], service);

                service = strtok(NULL, del\_serv);

                num++;

            }

            for (int j = 0; j < num; j++) {

                const char\* del\_serv = "=";

                char\* field = strtok(single\_fld[j], del\_serv);

                char\* value = strtok(NULL, del\_serv);

                if (strcmp(field, "last\_name") == 0)

                    strcpy(hd->data.last\_name, value);

                if (strcmp(field, "first\_name") == 0)

                    strcpy(hd->data.first\_name, value);

                if (strcmp(field, "middle\_name") == 0)

                    strcpy(hd->data.middle\_name, value);

                if (strcmp(field, "number") == 0)

                    strcpy(hd->data.number, value);

                if (strcmp(field, "bonus\_id") == 0)

                    hd->data.bonus\_id = atoi(value);

                if (strcmp(field, "discount\_id") == 0)

                    hd->data.discount\_id = atoi(value);

            }

        }

        i++;

        hd = hd->next;

    }

    return res;

}

Main.c:

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include "mem\_calc.h"

#include "list.h"

#include "select.h"

#include "read.h"

#include "filter.h"

struct node\* read\_db() {

    FILE\* file = fopen("DB.csv", "r");

    if (!file) {

        printf("Error opening file\n");

        return NULL;

    }

    char buffer[1024];

    struct row clients[100];

    int num\_clients = 0;

    int db\_size = -1;

    while (fgets(buffer, sizeof(buffer), file)) {

        db\_size++;

    }

    printf("число строк %d\n", db\_size);

    fclose(file);

    file = fopen("DB.csv", "r");

    // Чтение заголовка

    fgets(buffer, sizeof(buffer), file);

    struct node\* head = NULL;

    // Чтение строк

    while (fgets(buffer, sizeof(buffer), file)) {

        // Разбиение строки на поля

        const char\* delim = ";";

        char\* last\_name = strtok(buffer, delim);

        char\* first\_name = strtok(NULL, delim);

        char\* middle\_name = strtok(NULL, delim);

        char\* number\_str = strtok(NULL, delim);

        char\* discount\_str = strtok(NULL, delim);

        char\* bonus\_str = strtok(NULL, delim);

        char\* services\_str = strtok(NULL, "\n");

        // Заполнение структуры клиента

        if (1 || last\_name && first\_name && middle\_name && number\_str && services\_str)

        {

            strcpy(clients[num\_clients].last\_name, last\_name);

            strcpy(clients[num\_clients].first\_name, first\_name);

            strcpy(clients[num\_clients].middle\_name, middle\_name);

            strcpy(clients[num\_clients].number, number\_str);

            clients[num\_clients].discount\_id = atoi(discount\_str);

            clients[num\_clients].bonus\_id  = atoi(bonus\_str);

            // Разбиение списка услуг на отдельные названия

            const char\* del\_serv = " ";

            char\* service = strtok(services\_str, del\_serv);

            int i = 0;

            while (service && i < MAX\_SERVICES) {

                strncpy(clients[num\_clients].services[i], service, SERVICE\_NAME\_LENGTH);

                service = strtok(NULL, del\_serv);

                i++;

            }

            insert\_back(&head, clients[num\_clients]);

            num\_clients++;

        }

    }

    fclose(file);

    return head;

}

void main()

{

    struct node\* head = read\_db();

    struct row temp = {"last", "FIRST", "middle", "89173456789", {"s1","s2","s333"}};

    insert\_back(&head, temp);

    print\_list(head);

    printf("\n\nBefore processing\n");

    head = read\_command(head);

    printf("\n\nAfter processing\n");

    print\_list(head);

    free\_list(head);

    print\_mem();

    return;

}