Министерство науки и высшего образования Российской Федерации

Федеральное государственное автономное образовательное учреждение высшего образования

«Санкт-Петербургский политехнический университет Петра Великого»

—

Институт компьютерных наук и кибербезопасности

**Высшая школа кибербезопасности**

**ЛАБОРАТОРНАЯ РАБОТА № 3**

**Записная книжка**

по дисциплине «Структуры данных»

Выполнили

студенты гр. 5131001/30002 Мишенев Н. С.

Квашенникова В. М.

<*подпись*>

Руководитель

программист Вагисаров В. Б.

<*подпись*>

Санкт-Петербург

2024г.

**СОДЕРЖАНИЕ**

[ЦЕЛЬ РАБОТЫ 1](#_Toc1605669050)

[ХОД РАБОТЫ 1](#_Toc1993726051)

[ОТВЕТЫ НА КОНТРОЛЬНЫЕ ВОПРОСЫ 1](#_Toc1053662956)

[ВЫВОД 1](#_Toc1470412282)

[ПРИЛОЖЕНИЕ А 1](#_Toc1275815085)

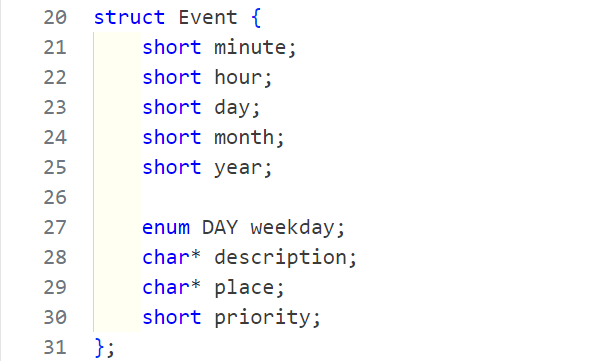
**ЦЕЛЬ РАБОТЫ**

Необходимо разработать программу, реализующую функционал записной книжки дел:  
1) хранение объектов типа "событие":  
 - год, месяц, число, час, минута события  
 - день недели  
 - описание события (текст)  
 - место события (текст)  
 - важность (0-10)  
   
2) возможность добавления новых событий  
3) возможность удаления событий  
4) вывод всех событий на экран:  
 - возможность сортировки событий по дате  
 - возможность сортировки событий по важности  
 - возможность фильтрации событий по месту  
5) возможность сохранения и загрузки событий из файла (одного и того же)  
   
Требования:   
 - должна быть реализована "защита от дурака", которая не позволит вводить некорректные данные в поля  
 - день недели должен высчитываться автоматически  
 - хранение событий должно быть реализовано на основе деревьев бинарного поиска  
 - сортировки должны быть реализованы с помощью обхода деревьев  
 - фильтр должен работать в том числе и по подстрокам  
 - структура хранения событий должна быть оптимальна по логике и памяти

**ХОД РАБОТЫ**

1. **Реализация хранения данных типа “событие”**

Для того чтобы работать с наборами данных, составляя их них события, была создана структура ***Event.*** Разработанная структура имеет следующий набор полей:



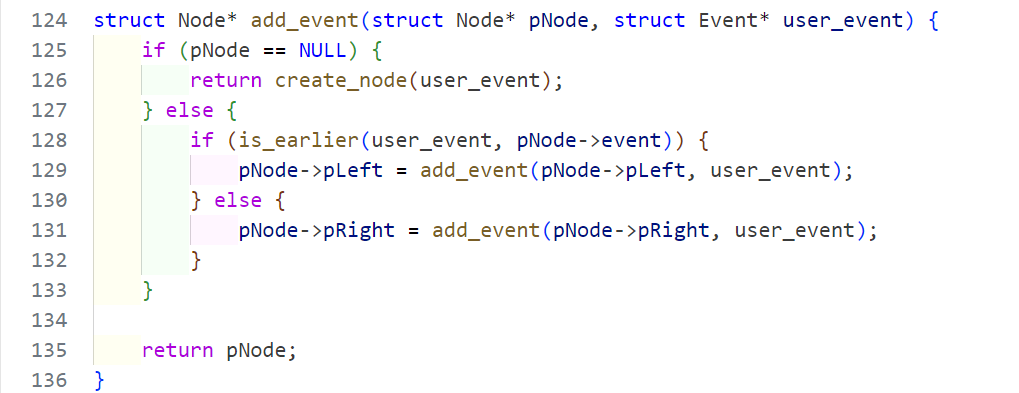
*Рис. 1. Объявление структуры “События”*

Данные о точном времени созданного события (минута, час, день, месяц, год). День недели, который будет вычисляться с помощью ранее полученных данных о времени события. Описание события и место его проведения (например адрес проведения). Приоритет события по 10-ти бальной шкале.

Такой набор полей данной структуры, позволяет практически однозначно определить место события в дереве поиска, а также оптимально выделить память для каждого объекта такого типа.

1. **Реализация функционала добавления события**

Так как для оптимального хранения и возможности быстрого поиска среди всех таких событий было решено использовать бинарное дерево поиска, то функционал добавления события основан на логике добавления каких-либо данных в такую структуру.



*Рис. 2. Реализация функционала добавления события*

Представленная функция ***add\_event()***, рекурсивно проходит всё дерево, пока не найдет подходящее место для нового объекта “события”. Функция, используемая для выбора дальнейшей ветки прохода, ***is\_earlier()***, проверяет, какое событие произойдет или произошло раньше. Если же события имеют идентичный набор данных о времени, то “ранним” событием, будет считаться событие с большим приоритетом.

Если же в какой-то момент прохода по дереву, найдено подходящее, пустое место, то с помощью функции ***create\_node(),*** в дереве создастся ячейка, содержащая переданное событие.

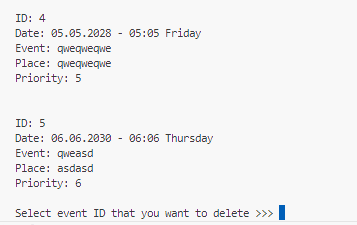
1. **Реализация функционала удаления события**

Реализация функции удаления события, основана на логике удаления ячейки дерева, содержащей это событие.



*Рис. 3. Реализация функционала удаления события*

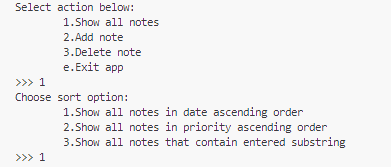
Схоже с реализацией добавления ячейки, мы должны сначала найти интересующую нас ячейку. Для этого, после вызова функции удаления события, всем ячейкам присваивается ID относительно их порядка возрастания. ID ячейки генерируется только при удалении ячейки, так как используется только для единоразовой идентификации удаляемой ячейки.



*Рис. 4. использование ID при удалении события*

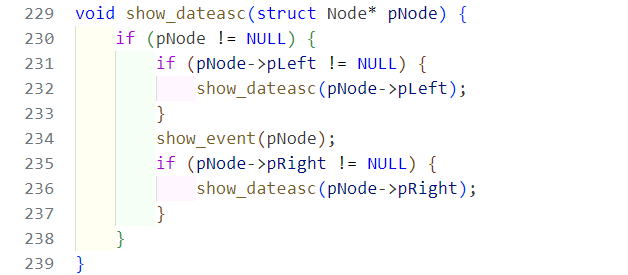
1. **Реализация вывода события на экран с использованием фильтрации и сортировок по полям события**

Для того чтобы сортировать выводимые события, используется инфиксный обход дерева, так как он позволяет просмотреть вся ячейки дерева в порядке “возрастания” даты события.



*Рис. 5. Меню выбора режима вывода событий на экран*

Для примера реализации инфиксного обхода приведем, к примеру одну из функций обхода.



*Рис. 6. Функция вывода событий в порядке возрастания даты*

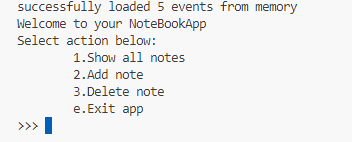
Данный обход осуществляется рекурсивным образом, и позволяет нам, как угодно, фильтровать данные при обходе. Функция ***show\_event()*** форматирует данные из структуры события для их последующего вывода на экран.

Таким же образом реализована сортировка по приоритетам событий. Запускается цикл для переменной ***i*** от 1 до 10, внутри которого, все события, которые имеют ***i*** приоритет, выводятся на экран.

При поиске по подстроке, текущее событие выводится только при наличии введенной подстроки в поле места структуры события. Так как при этом используется инфиксный обход, то временной порядок выводимых событий всегда сохраняется.

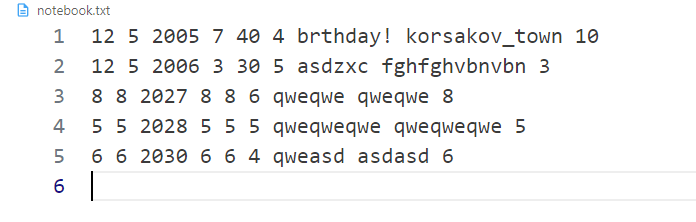
1. **Реализация хранения, загрузки и записи событий в файл**

При каждом запуске программы производится проверка существования файла, для хранения событий. Если файла не существует, то он создается, а если же существует, то выделяется необходимое количество памяти для создания объектов событий и они по очереди добавляются в текущий экземпляр дерева и об этом сообщается пользователю.



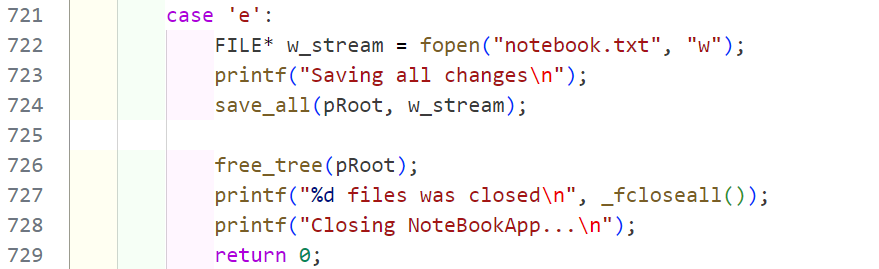
*Рис. 7. Загрузка событий из памяти*

Однако, для того чтобы можно было считывать из файла все эти события, их нужно структурировать особым образом и записать в файл.



*Рис. 8. Хранение событий в файле*

Для этого, перед каждым закрытием программы, производится сохранение текущего экземпляра дерева в файл. Для этого открывается поток записи в файл, и, с помощью инфиксного обхода дерева, все события по очереди записываются в файл, сохраняя временной порядок между друг другом.



*Рис. 9. Сохранение событий при закрытии*

1. **Реализация обязательных требований.**

Также, в ходе создания программы, были реализованы дополнительные обязательные требования. Одним из таких требований была реализация “защиты от дурака”, которая бы не позволяла пользователю вводить некорректные данные.

Для этого получение данных для полей события осуществляется через дополнительные функции.

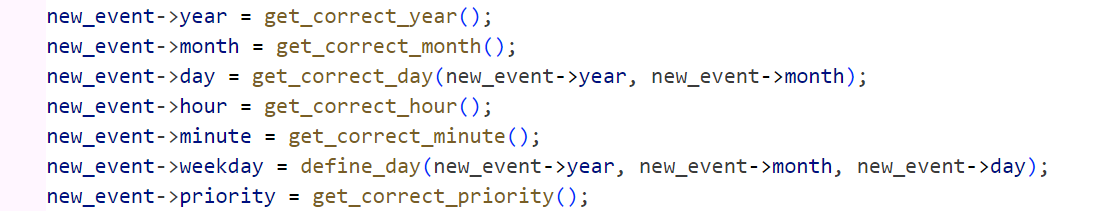
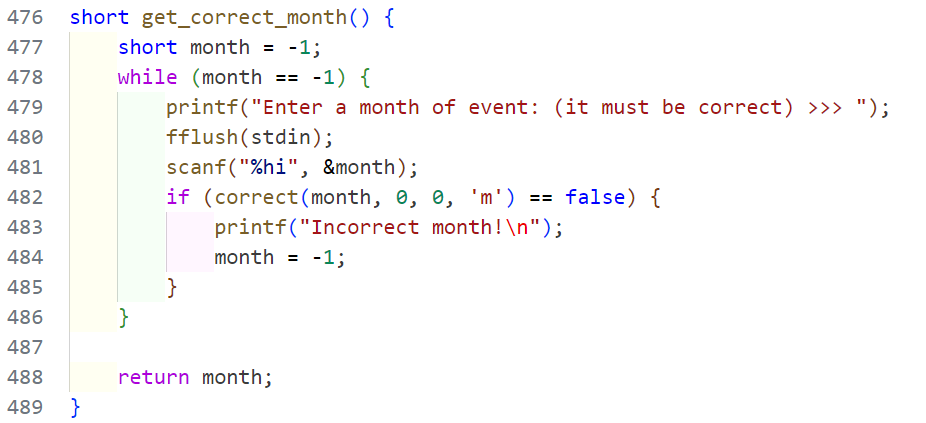


Рис. 10. Получение данных “события”

Внутри каждой из этих функций данные прогоняются через валидатор, который или принимает их, или бракует, в таком случае, пользователь получает сообщение об ошибке и ему предлагается попытаться ввести данные снова



*Рис. 11. Реализация защиты от дурака*

Также, день недели вычисляется автоматически с помощью некоторой формулы, которая в свою очередь основывается на текущем месяце, годе и факте о високосности года.

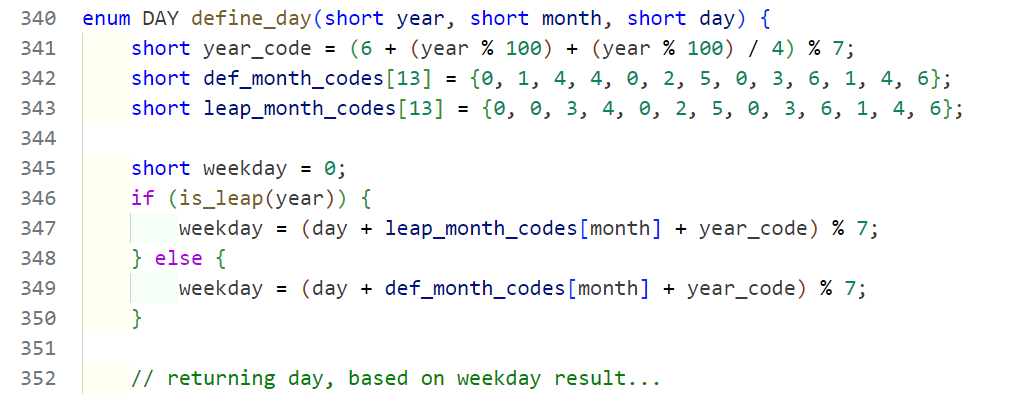


Рис. 12. Вычисление для недели

**ВЫВОД**

В ходе работы была реализована программа типа “Записная книжка” с возможностью сохранения событий в файл. Была изучена структура бинарного дерева поиска и основные операции над ним.

**ПРИЛОЖЕНИЕ А**

Листинг 1 - исходный код программы на языке С

#include <stdio.h>

#include <stdbool.h>

#include <stdlib.h>

#include <string.h>

#define FIELDMAXLEN 50

#define MAXEVENTLEN 22 + FIELDMAXLEN \* 2

#define MAXSUBSTRINGLEN 25

enum DAY {

MONDAY = 1,

TUESDAY = 2,

WEDNESDAY = 3,

THURSDAY = 4,

FRIDAY = 5,

SUNDAY = 6,

SATURDAY = 7

};

struct Event {

short minute;

short hour;

short day;

short month;

short year;

enum DAY weekday;

char\* description;

char\* place;

short priority;

};

struct Node {

short id;

struct Event\* event;

struct Node\* pRight;

struct Node\* pLeft;

};

void copy(char\* str1, char\* str2, int len) {

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

str1[i] = str2[i];

}

}

void strip(char\* str) {

int i = 0;

while (str[i] != '\0') {

if (str[i] == ' ') {

str[i] = '\_';

}

if (str[i] == '\n') {

str[i] = '\0';

return;

}

i++;

}

}

void clear(char\* str, int strlen) {

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

str[i] = '\0';

}

}

int is\_earlier(struct Event\* to\_put, struct Event\* curr) {

if (to\_put->year < curr->year) {

return 1;

}

if (to\_put->year > curr->year) {

return 0;

}

if (to\_put->year == curr->year) {

if (to\_put->month < curr->month) {

return 1;

}

if (to\_put->month > curr->month) {

return 0;

}

if (to\_put->month == curr->month) {

if (to\_put->day < curr->day) {

return 1;

}

if (to\_put->day > curr->day) {

return 0;

}

if (to\_put->day == curr->day) {

if (to\_put->hour < curr->hour) {

return 1;

}

if (to\_put->hour > curr->hour) {

return 0;

}

if (to\_put->hour == curr->hour) {

if (to\_put->minute < curr->minute) {

return 1;

}

if (to\_put->minute > curr->minute) {

return 0;

}

if (to\_put->minute == curr->minute) {

return (to\_put->priority > curr->priority) ? 1 : 0;

}

}

}

}

}

return 0;

}

struct Node\* create\_node(struct Event\* user\_event) {

struct Node\* pNode = (struct Node\*)malloc(sizeof(struct Node));

if (pNode == NULL) {

printf("error occured while allocating memory");

return NULL;

}

pNode->id = -1;

pNode->event = user\_event;

pNode->pRight = NULL;

pNode->pLeft = NULL;

return pNode;

}

struct Node\* add\_event(struct Node\* pNode, struct Event\* user\_event) {

if (pNode == NULL) {

return create\_node(user\_event);

} else {

if (is\_earlier(user\_event, pNode->event)) {

pNode->pLeft = add\_event(pNode->pLeft, user\_event);

} else {

pNode->pRight = add\_event(pNode->pRight, user\_event);

}

}

return pNode;

}

void show\_event(struct Node\* pNode) {

if (pNode->event == NULL || pNode == NULL) {

printf("Nothing to show!\n");

return;

}

char weekday[10] = { 0 };

switch (pNode->event->weekday) {

case 1:

copy(weekday, "Monday", 7);

break;

case 2:

copy(weekday, "Tuesday", 8);

break;

case 3:

copy(weekday, "Wednesday", 10);

break;

case 4:

copy(weekday, "Thursday", 9);

break;

case 5:

copy(weekday, "Friday", 7);

break;

case 6:

copy(weekday, "Sunday", 7);

break;

case 7:

copy(weekday, "Saturday", 9);

break;

default:

printf("Unknown error occured in show\_event()!\n");

return;

}

char day\_zero = ' ';

char month\_zero = ' ';

char hour\_zero = ' ';

char minute\_zero = ' ';

if (pNode->event->day < 10) {

day\_zero = '0';

}

if (pNode->event->month < 10) {

month\_zero = '0';

}

if (pNode->event->hour < 10) {

hour\_zero = '0';

}

if (pNode->event->minute < 10) {

minute\_zero = '0';

}

printf("\nID: %hi\nDate: %c%hi.%c%hi.%hi - %c%hi:%c%hi %s\nEvent: %s\nPlace: %s\nPriority: %hi\n\n", \

pNode->id,

day\_zero,

pNode->event->day,

month\_zero,

pNode->event->month,

pNode->event->year,

hour\_zero,

pNode->event->hour,

minute\_zero,

pNode->event->minute,

weekday,

pNode->event->description,

pNode->event->place,

pNode->event->priority

);

return;

}

bool found\_substring(char\* string, char\* substring) {

bool substring\_found = false;

int correct\_chars = 0;

for (int i = 0; i < (strlen(string) - strlen(substring) + 1); i++) {

correct\_chars = 0;

for (int j = 0; j < strlen(substring); j++) {

if (string[i+j] == substring[j]) {

correct\_chars++;

}

}

if (correct\_chars == strlen(substring)) {

return true;

}

}

return false;

}

void show\_dateasc(struct Node\* pNode) {

if (pNode != NULL) {

if (pNode->pLeft != NULL) {

show\_dateasc(pNode->pLeft);

}

show\_event(pNode);

if (pNode->pRight != NULL) {

show\_dateasc(pNode->pRight);

}

}

}

void show\_priorityasc(struct Node\* pNode, int\* priority) {

if (pNode != NULL) {

if (pNode->pLeft != NULL) {

show\_priorityasc(pNode->pLeft, priority);

}

if (pNode->event->priority == \*priority) {

show\_event(pNode);

}

if (pNode->pRight != NULL) {

show\_priorityasc(pNode->pRight, priority);

}

}

}

void show\_substring(struct Node \*pNode, char\* substring) {

if (pNode != NULL) {

if (pNode->pLeft != NULL) {

show\_substring(pNode->pLeft, substring);

}

if (found\_substring(pNode->event->place, substring) == true) {

show\_event(pNode);

}

if (pNode->pRight != NULL) {

show\_substring(pNode->pRight, substring);

}

}

}

void count\_events(struct Node\* pNode, int\* count) {

if (pNode != NULL) {

if (pNode->pLeft != NULL) {

count\_events(pNode->pLeft, count);

}

\*count++;

if (pNode->pRight != NULL) {

count\_events(pNode->pRight, count);

}

}

}

void update\_tree(struct Node\* pNode, int\* id) {

if (pNode != NULL) {

if (pNode->pLeft != NULL) {

update\_tree(pNode->pLeft, id);

}

pNode->id = \*id;

(\*id)++;

if (pNode->pRight != NULL) {

update\_tree(pNode->pRight, id);

}

}

}

void save\_all(struct Node\* pNode, FILE\* stream) {

if (pNode != NULL) {

if (pNode->pLeft != NULL) {

save\_all(pNode->pLeft, stream);

}

fprintf(stream, "%hi %hi %hi %hi %hi %hi %s %s %hi\n", \

pNode->event->day,

pNode->event->month,

pNode->event->year,

pNode->event->hour,

pNode->event->minute,

pNode->event->weekday,

pNode->event->description,

pNode->event->place,

pNode->event->priority

);

if (pNode->pRight != NULL) {

save\_all(pNode->pRight, stream);

}

}

}

void free\_tree(struct Node\* pNode) {

if (pNode != NULL) {

if (pNode->pLeft != NULL) {

free\_tree(pNode->pLeft);

}

if (pNode->pLeft == NULL && pNode->pRight == NULL) {

free(pNode);

pNode == NULL;

}

if (pNode->pRight != NULL) {

free\_tree(pNode->pRight);

}

}

}

bool is\_leap(short year) {

if (year == 2000 || year % 4 == 0) {

return true;

} else {

return false;

}

}

enum DAY define\_day(short year, short month, short day) {

short year\_code = (6 + (year % 100) + (year % 100) / 4) % 7;

short def\_month\_codes[13] = {0, 1, 4, 4, 0, 2, 5, 0, 3, 6, 1, 4, 6};

short leap\_month\_codes[13] = {0, 0, 3, 4, 0, 2, 5, 0, 3, 6, 1, 4, 6};

short weekday = 0;

if (is\_leap(year)) {

weekday = (day + leap\_month\_codes[month] + year\_code) % 7;

} else {

weekday = (day + def\_month\_codes[month] + year\_code) % 7;

}

switch (weekday) {

case 0:

return SATURDAY;

case 1:

return SUNDAY;

case 2:

return MONDAY;

case 3:

return TUESDAY;

case 4:

return WEDNESDAY;

case 5:

return THURSDAY;

case 6:

return FRIDAY;

default:

printf("Unknown error occured in define\_day()");

break;

}

}

bool correct(short data, short year, short month, char mode) {

short days[13] = {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};

switch (mode) {

case 'y':

return (data > 1900 && data < 2100) ? true : false;

case 'm':

return (data >= 1 && data <= 12) ? true : false;

case 'd':

if (data > 0 && data <= days[month]) {

if (month == 2 && is\_leap(year) == true) {

return data <= 29 ? true : false;

} else if (month == 2 && is\_leap(year) == false) {

return data <= 28 ? true : false;

} else {

return true;

}

} else {

return false;

}

case 'h':

return (data >= 0 && data <= 24) ? true : false;

case 'i':

return (data >= 0 && data <= 60) ? true : false;

case 'p':

return (data >= 0 && data <= 10) ? true : false;

default:

printf("Unknown error occured in correct()!");

break;

}

}

struct Node\* find\_earliest(struct Node\* pNode) {

if (pNode == NULL) {

return NULL;

} else if (pNode->pLeft != NULL) {

return find\_earliest(pNode->pLeft);

}

return pNode;

}

struct Node\* delete\_node(struct Node\* pRoot, int id) {

if (pRoot == NULL)

return NULL;

if (id > pRoot->id) {

pRoot->pRight = delete\_node(pRoot->pRight, id);

} else if (id < pRoot->id) {

pRoot->pLeft = delete\_node(pRoot->pLeft, id);

} else {

if (pRoot->pLeft == NULL && pRoot->pRight == NULL) {

free(pRoot);

return NULL;

} else if (pRoot->pLeft == NULL || pRoot->pRight == NULL) {

struct Node\* temp\_node;

if (pRoot->pLeft == NULL) {

temp\_node = pRoot->pRight;

} else {

temp\_node = pRoot->pLeft;

}

free(pRoot);

return temp\_node;

} else {

struct Node\* temp\_node = find\_earliest(pRoot->pRight);

pRoot->event = temp\_node->event;

pRoot->pRight = delete\_node(pRoot->pRight, temp\_node->id);

}

}

return pRoot;

}

short get\_correct\_priority() {

short priority = -1;

while (priority == -1) {

printf("Enter a priority of event: (it must be between 0 and 10) >>> ");

fflush(stdin);

scanf("%hi", &priority);

if (correct(priority, 0, 0, 'p') == false) {

printf("Incorrect priority!\n");

priority = -1;

}

}

return priority;

}

short get\_correct\_year() {

short year = -1;

while (year == -1) {

printf("Enter a year of event: (it must be between 1901 and 2099) >>> ");

fflush(stdin);

scanf("%hi", &year);

if (correct(year, 0, 0, 'y') == false) {

printf("Incorrect year!\n");

year = -1;

}

}

return year;

}

short get\_correct\_month() {

short month = -1;

while (month == -1) {

printf("Enter a month of event: (it must be correct) >>> ");

fflush(stdin);

scanf("%hi", &month);

if (correct(month, 0, 0, 'm') == false) {

printf("Incorrect month!\n");

month = -1;

}

}

return month;

}

short get\_correct\_day(short year, short month) {

short day = -1;

while (day == -1) {

printf("Enter a day of event: (it must be correct) >>> ");

fflush(stdin);

scanf("%hi", &day);

if (correct(day, year, month, 'd') == false) {

printf("Incorrect day!\n");

day = -1;

}

}

return day;

}

short get\_correct\_hour() {

short hour = -1;

while (hour == -1) {

printf("Enter a hour of event: (it must be correct) >>> ");

fflush(stdin);

scanf("%hi", &hour);

if (correct(hour, 0, 0, 'h') == false) {

printf("Incorrect hour!\n");

hour = -1;

}

}

return hour;

}

short get\_correct\_minute() {

short minute = -1;

while (minute == -1) {

printf("Enter a minute of event: (it must be correct) >>> ");

fflush(stdin);

scanf("%hi", &minute);

if (correct(minute, 0, 0, 'i') == false) {

printf("Incorrect minute!\n");

minute = -1;

}

}

return minute;

}

char\* get\_correct\_description() {

char\* description = (char\*)calloc(FIELDMAXLEN, sizeof(char));

description[0] == '\0';

while (description[0] == '\0') {

printf("Enter a description of event: (it must be < 100 characters long) >>> ");

fflush(stdin);

fgets(description, FIELDMAXLEN, stdin);

if (strlen(description) > FIELDMAXLEN) {

printf("Incorrect description!\n");

description[0] == '\0';

}

}

strip(description);

return description;

}

char\* get\_correct\_place() {

char\* place = (char\*)calloc(FIELDMAXLEN, sizeof(char));

place[0] == '\0';

while (place[0] == '\0') {

printf("Enter a place of event: (it must be < 100 characters long) >>> ");

fflush(stdin);

fgets(place, FIELDMAXLEN, stdin);

if (strlen(place) > FIELDMAXLEN) {

printf("Incorrect place!\n");

place[0] == '\0';

}

}

strip(place);

return place;

}

int main(void) {

FILE\* r\_count\_stream = fopen("notebook.txt", "a+");

int events\_count = 0;

char c;

bool no\_events;

while ((c = fgetc(r\_count\_stream)) != EOF) {

if (c == '\n') {

events\_count++;

}

}

fclose(r\_count\_stream);

FILE\* r\_stream = fopen("notebook.txt", "r");

fseek(r\_stream, 0L, SEEK\_SET);

struct Node\* pRoot = NULL;

int loaded = 0;

for (int i = 0; i < events\_count; i++) {

struct Event\* read\_event = (struct Event\*)malloc(sizeof(struct Event));

if (read\_event == NULL) {

printf("Error occured while allocating memory");

}

read\_event->description = (char\*)malloc(sizeof(char) \* FIELDMAXLEN);

if (read\_event->description == NULL) {

printf("Error occured while allocating memory");

}

read\_event->place = (char\*)malloc(sizeof(char) \* FIELDMAXLEN);

if (read\_event->place == NULL) {

printf("Error occured while allocating memory");

}

int read = fscanf(r\_stream, "%hi %hi %hi %hi %hi %hi %s %s %hi\n", \

&(read\_event->day),

&(read\_event->month),

&(read\_event->year),

&(read\_event->hour),

&(read\_event->minute),

&(read\_event->weekday),

read\_event->description,

read\_event->place,

&(read\_event->priority)

);

if (read == 9) {

loaded++;

}

if (pRoot == NULL) {

pRoot = add\_event(pRoot, read\_event);

} else {

add\_event(pRoot, read\_event);

}

}

if (loaded != 0) {

printf("successfully loaded %d events from memory\n", loaded);

}

char user\_input = ' ';

printf("Welcome to your NoteBookApp\n");

while (1) {

printf("Select action below:\n\t1.Show all notes\n\t2.Add note\n\t3.Delete note\n\te.Exit app\n>>> ");

fflush(stdin);

scanf("%c", &user\_input);

switch(user\_input) {

case '1':

if (pRoot == NULL) {

printf("No events in NoteBook\n");

} else {

printf("Choose sort option:\n\t1.Show all notes in date ascending order\n\t2.Show all notes in priority ascending order\n\t3.Show all notes that contain entered substring\n>>> ");

char sort\_option = ' ';

fflush(stdin);

scanf("%c", &sort\_option);

if (sort\_option == '1') {

show\_dateasc(pRoot);

} else if (sort\_option == '2') {

for (int i = 0; i <= 10; i++) {

show\_priorityasc(pRoot, &i);

}

} else if (sort\_option == '3') {

char substring[MAXSUBSTRINGLEN] = { 0 };

printf("Enter substring to search in place field >>> ");

scanf("%s", substring);

printf("Substring was found in these events:\n");

show\_substring(pRoot, substring);

} else {

printf("Error input detected!");

}

}

break;

case '2':

struct Event\* new\_event = (struct Event\*)malloc(sizeof(struct Event));

if (new\_event == NULL) {

printf("Error occured while allocating memory");

}

new\_event->year = get\_correct\_year();

new\_event->month = get\_correct\_month();

new\_event->day = get\_correct\_day(new\_event->year, new\_event->month);

new\_event->hour = get\_correct\_hour();

new\_event->minute = get\_correct\_minute();

new\_event->weekday = define\_day(new\_event->year, new\_event->month, new\_event->day);

new\_event->priority = get\_correct\_priority();

new\_event->description = (char\*)malloc(sizeof(char) \* FIELDMAXLEN);

if (new\_event->description == NULL) {

printf("Error occured while allocating memory");

}

copy(new\_event->description, get\_correct\_description(), FIELDMAXLEN);

new\_event->place = (char\*)malloc(sizeof(char) \* FIELDMAXLEN);

if (new\_event->place == NULL) {

printf("Error occured while allocating memory");

}

copy(new\_event->place, get\_correct\_place(), FIELDMAXLEN);

if (pRoot == NULL) {

pRoot = add\_event(pRoot, new\_event);

} else {

add\_event(pRoot, new\_event);

}

printf("Event successfully added!\n");

break;

case '3':

if (pRoot == NULL) {

printf("No events in NoteBook\n");

} else {

int id = 1;

int choosen\_id = 0;

printf("All events: ");

update\_tree(pRoot, &id);

show\_dateasc(pRoot);

printf("Select event ID that you want to delete >>> ");

fflush(stdin);

scanf("%d", &choosen\_id);

delete\_node(pRoot, choosen\_id);

}

break;

case 'e':

FILE\* w\_stream = fopen("notebook.txt", "w");

printf("Saving all changes\n");

save\_all(pRoot, w\_stream);

free\_tree(pRoot);

printf("%d files was closed\n", \_fcloseall());

printf("Closing NoteBookApp...\n");

return 0;

default:

printf("Detected input error!\n");

}

}

}