МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

Інститут **ІКНІ**
Кафедра **ПЗ**

ЗВІТ

До лабораторної роботи № 10
На тему: *"динамічні структури даних"*
З дисципліни: *"Основи програмування"*

Лектор:
ст.викл. каф. ПЗ
Муха Т.О.

Виконав:
ст. гр. ПЗ-15
Марущак А.С.

Прийняла:
асист. каф. ПЗ
Заводовська Н.О.

« ____ » _____ 2021 р.

Σ= ____ .

Львів – 2021

Тема роботи: динамічні структури даних.
Мета роботи: оволодіти практичними прийомами створення та опрацювання динамічних списків.

Теоретичні відомості

Динамічні структури даних характеризуються відсутністю фізичної суміжності елементів структури в пам'яті (різні елементи «розкидані» в пам'яті), а також непостійністю та непередбачуваністю розміру (в процесі виконання програми можуть додаватися нові елементи чи видалятися існуючі). Динамічні структури являють собою сукупність об’єктів (як правило, розміщених у динамічній пам'яті), кожен з яких містить інформацію про адресу в пам'яті іншого об’єкта, зв’язаного з ним.

Оскільки елементи динамічної структури розміщуються за непередбачуваними адресами пам'яті, то адресу кожного елемента структури неможливо обчислити за адресою попереднього або наступного елемента. Між елементами динамічної структури встановлюються явні зв’язки за допомогою вказівників. Таке представлення даних у пам'яті називається зв’язним. Елемент динамічної структури складається з двох полів:

- інформаційного поля (поля даних), яке містить ті дані, для яких власне і створюється структура;
- поле зв'язків, в якому містяться один або декілька вказівників, що зв’язують даний елемент з іншими елементами структури.

Інформаційне поле може бути представлене не лише сукупністю даних вбудованих типів, але й масивами, структурами тощо.

Розмір динамічної структури обмежується лише доступним обсягом машинної пам'яті. При зміні логічної послідовності елементів структури вимагається не переміщення даних у пам'яті, а лише корекція вказівників.

Однак, робота зі вказівниками вимагає більшої кваліфікації програміста, для зберігання вказівників витрачається додаткова пам'ять, а доступ до елементів зв’язної структури може бути менш ефективним за часом. Для порівняння: при суміжному розміщенні однорідних даних у пам'яті (яке має місце в масиві) для обчислення адреси будь-якого елемента потрібно знати номер цього елемента та адреси початку області пам'яті (доступ прямий), а для доступу до елемента динамічної структури потрібно послідовно «перебрати» усі елементи динамічної структури від початку.

Список – це впорядкована множина, що складається зі змінного числа елементів, до яких застосовні операції додавання та видалення елементів. Якщо вказівник посилається лише на один інший елемент списку, то такий список називається однонапрямленим. Наприклад, елемент однонапрямленого списку може описуватися структурою:

```
struct Item { int num; struct Item* next; };
```

Якщо елемент посилається і на попередній, і на наступний елемент, то такий список є двонапрямленим. Якщо вказівник в останньому елементі не

встановлений у NULL, а посилається на перший елемент списку, то такий список називається кільцевим.

ЛАБОРАТОРНЕ ЗАВДАННЯ

1. Ознайомитися з теоретичним матеріалом викладеним вище в даній інструкції і виконати приклади програм.
2. Одержати індивідуальне завдання.
3. Скласти програму на мові С++ у відповідності з розробленим алгоритмом.
4. Виконати обчислення по програмі.
5. Підготувати та здати звіт про виконання лабораторної роботи.

Індивідуальне завдання

Виконати завдання з лабораторної роботи № 8, організувавши послідовність структур в однозв'язний список. Реалізувати операцію вставки нового елемента у відсортований список і операцію вилучення зі списку даних, які відповідають одній з наступних умов:

15) про книги з вартістю менше 5 грв.;

Завдання з ЛР №8:

26. Відсортувавати за прізвищем в алфавітному порядку дані про книги, вартість яких менша середньої в бібліотеці.

Код програми:
(lab10_lib.h):

```
#ifndef LAB10
#define LAB10

#include <stdarg.h>
#include <stddef.h>

#define max(a, b) a > b ? a : b

#define REPEAT_CHAR(ch, i) \
    for(int z = 0; z < (i); z++){\
        printf("%c", ch);\
    }\

#define MAKE_LINE(plainCh, nodeCh, autS, titS, yearS, pageS, priceS)\
    printf("%c", nodeCh);\
    REPEAT_CHAR(plainCh, (autS));\
    printf("%c", nodeCh);\
    REPEAT_CHAR(plainCh, (titS));\
    printf("%c", nodeCh);\
    REPEAT_CHAR(plainCh, (yearS));\
    printf("%c", nodeCh);\
    REPEAT_CHAR(plainCh, (pageS));\
    printf("%c", nodeCh);\
    REPEAT_CHAR(plainCh, (priceS));\
    printf("%c\n", nodeCh);


#define MAX_NAME_LEN 20
#define MAX_SURNAME_LEN 20
```

```
#define MAX_COMMAND_LEN 20
#define MAX_ARGS_COUNT 20
#define MAX_DESCRIPTION_LEN 1024
#define MAX_LISTNAME_LEN 20
#define MAX_TITLE_LEN 50
#define MAX_LINE_LEN 256

typedef struct Book
{
    char title[MAX_TITLE_LEN];
    struct Author
    {
        char name[MAX_NAME_LEN];
        char surname[MAX_SURNAME_LEN];
    }   author;
    double price;
    int pubYear;
    int pageCount;
} Book;

typedef struct ListItem{
    Book data;
    struct ListItem *next;
} ListItem;

typedef struct List{
    char name[MAX_LISTNAME_LEN];
    ListItem *head;
} List;

typedef struct CLArgs{
    int argc;
    char *argv[MAX_ARGS_COUNT];
} CLArgs;

typedef struct Command{
    char commandName[MAX_COMMAND_LEN];
    char commandDesc[MAX_DESCRIPTION_LEN];
    void (*commandAction)(const CLArgs *const args);
} Command;

typedef int (*BookComparer)(Book a, Book b);
typedef int (*BookPredicate)(Book a, va_list va);
typedef double (*BookSelector)(Book a, va_list va);

CLArgs* parseCommandLine(char *str, char **command);
void executeLine(char *line);
void executeFile(char *fileName);
void launchConsole();
void cls();

#endif
```

(lab10_lib.c):

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <ctype.h>
#include "lab10_lib.h"

#pragma region Utils

int numLen(int num)
```

```
{
    int result = 0;
    while (num > 0)
    {
        num /= 10;
        result++;
    }
    return result;
}

int getAuthorFullNameLen(const Book book)
{
    return strlen(book.author.name) + strlen(book.author.surname) + 1;
}

int longestAuthorFieldSize(const List *const books)
{
    if (!books)
        return 0;
    ListItem *curr = books->head;
    int result = 0;
    while (curr)
    {
        int currLen = getAuthorFullNameLen(curr->data);
        result = currLen > result ? currLen : result;
        curr = curr->next;
    }
    return result;
}

int longestBookTitleSize(const List *const books)
{
    if (!books)
        return 0;
    ListItem *curr = books->head;
    int result = 0;
    while (curr)
    {
        int currLen = strlen(curr->data.title);
        if (currLen > result)
            result = currLen;
        curr = curr->next;
    }
    return result;
}

void longestNumberFieldsSize(const List *const books, int *const yearSize, int *const
pageSize, int *const priceSize)
{
    if (!books)
    {
        *yearSize = 0;
        *pageSize = 0;
        *priceSize = 0;
        return;
    }

    int
        maxYearLen = 0,
        maxPageLen = 0,
        maxPriceLen = 0,
        yearLen = 0,
        pageLen = 0,
        priceLen = 0;

    ListItem *curr = books->head;
```

```
    while (curr)
    {
        yearLen = numLen(curr->data.pubYear);
        pageLen = numLen(curr->data.pageCount);
        priceLen = numLen((int)(curr->data.price)) + 3;

        if (yearLen > maxYearLen)
            maxYearLen = yearLen;
        if (pageLen > maxPageLen)
            maxPageLen = pageLen;
        if (priceLen > maxPriceLen)
            maxPriceLen = priceLen;
        curr = curr->next;
    }

    *yearSize = maxYearLen;
    *pageSize = maxPageLen;
    *priceSize = maxPriceLen;
}

void getSizes(const List *const books, int *const authorColumnWidth, int *const
titleColumnWidth, int *const yearColumnWidth, int *const pageColumnWidth, int *const
priceColumnWidth)
{

    *authorColumnWidth = max(longestAuthorFieldSize(books), strlen("Author"));
    *titleColumnWidth = max(longestBookTitleSize(books), strlen("Title"));
    longestNumberFieldsSize(books, yearColumnWidth, pageColumnWidth, priceColumnWidth);
    *yearColumnWidth = max(*yearColumnWidth, strlen("Pub. year"));
    *pageColumnWidth = max(*pageColumnWidth, strlen("Pages"));
    *priceColumnWidth = max(*priceColumnWidth, strlen("Price"));
}

char *rtrim(char *s)
{
    char *back = s + strlen(s);
    while (isspace(*--back))
        ;
    *(back + 1) = '\0';
    return s;
}

char *ltrim(char *s)
{
    while (isspace(*s))
        s++;
    return s;
}

char *trim(char *s)
{
    return ltrim(rtrim(s));
}

char *qtrim(char *s)
{
    char *last = s + strlen(s) - 1;
    if (*last == '"')
        *last = 0;
    return *s == '"' ? s + 1 : s;
}

void prompt(const char const *msg, char *const buffer)
{
    printf(msg);
```

```
    gets(buffer);
    fflush(stdin);
}

int fexists(const char *const fileName)
{
    FILE *f = fopen(fileName, "r");
    if (f)
    {
        fclose(f);
        return 1;
    }
    return 0;
}

Book strToBook(char str[])
{
    Book result;

    char *pPart = strtok(str, ",");
    sscanf(pPart, "%s %s", &result.author.name, &result.author.surname);

    pPart = strtok(NULL, ",");
    strcpy(result.title, pPart);

    pPart = strtok(NULL, ",");
    result.pubYear = atoi(pPart);

    pPart = strtok(NULL, ",");
    result.pageCount = atoi(pPart);

    pPart = strtok(NULL, ",");
    result.price = strtod(pPart, NULL);

    return result;
}

Book getBookFromConsole()
{
    char authorsName[MAX_NAME_LEN];
    char authorsSurname[MAX_SURNAME_LEN];
    char title[MAX_TITLE_LEN];
    int pubYear;
    int pageCount;
    double price;

    printf("Enter author(name surname): ");
    scanf("%s %s", authorsName, authorsSurname);
    fflush(stdin);
    printf("Enter book's title: ");
    gets(title);
    printf("Enter publication year: ");
    scanf("%d", &pubYear);
    printf("Enter page count: ");
    scanf("%d", &pageCount);
    printf("Enter price: ");
    scanf("%lf", &price);

    Book newBook;
    strcpy(newBook.title, trim(title));
    strcpy(newBook.author.name, authorsName);
    strcpy(newBook.author.surname, authorsSurname);
    newBook.price = price;
    newBook.pageCount = pageCount;
    newBook.pubYear = pubYear;
```

```
    return newBook;
}

int isNumber(char *str)
{
    str = trim(str);
    for (char *p = str; *p != 0; p++)
    {
        if (!isdigit(*p) && *p != '.')
            return 0;
    }
    return 1;
}

int startsWith(char *str, char *begining)
{
    if (strlen(begining) > strlen(str))
        return 0;
    for (int i = 0; i < strlen(begining); i++)
    {
        if (tolower(str[i]) != tolower(begining[i]))
            return 0;
    }
    return 1;
}

int rightFileFormat(char *fileName)
{
    char *extension = strrchr(fileName, '.');

    if (extension)
    {
        if (!strcmp(extension, ".lsexe"))
            return 1;
    }
    return 0;
}

int isCorrectScriptFile(char *fileName)
{
    return fexists(fileName) && rightFileFormat(fileName);
}

void cls()
{
    system("cls");
}

#pragma endregion

#pragma region Comparers

int authorsNameBookComparer(Book a, Book b)
{
    return strcmp(a.author.name, b.author.name);
}

int authorsSurnameBookComparer(Book a, Book b)
{
    return strcmp(a.author.surname, b.author.surname);
}

int titleBookComaprer(Book a, Book b)
{
    return strcmp(a.title, b.title);
}
```

```
int yearBookComparer(Book a, Book b)
{
    return a.pubYear == b.pubYear ? 0 : a.pubYear > b.pubYear ? 1
                                                              : -1;
}

int pageBookComparer(Book a, Book b)
{
    return a.pageCount == b.pageCount ? 0 : a.pageCount > b.pageCount ? 1
                                                                      : -1;
}

int priceBookComparer(Book a, Book b)
{
    return a.price == b.price ? 0 : a.price > b.price ? 1
                                                      : -1;
}

#pragma endregion

#pragma region FiltraionPredicatesAndSelectors

double GetBooksField(Book a, va_list va)
{
    char *field = va_arg(va, char *);
    if (!strcmp(field, "price"))
    {
        return a.price;
    }
    if (!strcmp(field, "pages"))
    {
        return (double)a.pageCount;
    }
    if (!strcmp(field, "year"))
    {
        return (double)a.pubYear;
    }
    return 0;
}

int lessThanS(Book a, va_list va)
{
    double value = GetBooksField(a, va);
    va_arg(va, char *);
    char *limStr = va_arg(va, char *);
    double lim = strtod(limStr, NULL);

    return value < lim;
}

int moreThanS(Book a, va_list va)
{
    double value = GetBooksField(a, va);
    va_arg(va, char *);
    char *limStr = va_arg(va, char *);
    double lim = strtod(limStr, NULL);

    return value > lim;
}

double getListAverage(List *list, BookSelector selector, ...)
{
    if (!list || !list->head)
        return 0;
    va_list va;
```

```
    va_start(va, selector);
    ListItem *iter = list->head;
    double res = 0;
    int count = 0;
    while (iter)
    {
        res += selector(iter->data, va);
        iter = iter->next;
        count++;
    }
    return res / count;
}

int lessThanD(Book a, va_list va)
{
    double value = GetBooksField(a, va);
    va_arg(va, char *);
    double lim = va_arg(va, double);

    return value < lim;
}

int moreThanD(Book a, va_list va)
{
    double value = GetBooksField(a, va);
    va_arg(va, char *);
    double lim = va_arg(va, double);

    return value > lim;
}

int startsWithS(Book a, va_list va)
{
    char *field = va_arg(va, char *);
    char *starting = va_arg(va, char *);
    char *str = "";
    if (!strcmp(field, "title"))
        str = a.title;
    else if (!strcmp(field, "name"))
        str = a.author.name;
    else if (!strcmp(field, "surname"))
        str = a.author.surname;

    return startsWith(str, starting);
}

int notStartsWithS(Book a, va_list va)
{
    return !startsWithS(a, va);
}

#pragma endregion

#pragma region CommandsDeclarations

void exitProgram(const CLArgs *const args);
void clearConsole(const CLArgs *const args);
void getHelp(const CLArgs *const args);
void createList(const CLArgs *const args);
void getLists(const CLArgs *const args);
void addElement(const CLArgs *const args);
void switchList(const CLArgs *const args);
void formTable(const CLArgs *const args);
void loadFromFile(const CLArgs *const args);
void countList(const CLArgs *const args);
void deleteList(const CLArgs *const args);
```

```
void sortList(const CLArgs *const args);
void filterList(const CLArgs *const args);
void limitList(const CLArgs *const args);
void insertElement(const CLArgs *const args);
void saveToFile(const CLArgs *const args);

#pragma endregion

#pragma region CommandsAndListsStuff

static int listCount = 0;
static List *lists = NULL;

static int currentListIndex = -1;
static List *currentList = NULL;

static Command commands[] = {

    {"exit",
     "Closes the console.\n\n"
     "Syntax: exit [<modifiers>]\n\n"
     "Modifiers:\n\n"
     "\t-s, --success - to set exit code to SUCCESS(default).\n"
     "\t-f, --fail - to set exit code to FAILURE.\n\n",
     exitProgram},

    {"cls",
     "Clears the console.\n\n",
     clearConsole},

    {"help",
     "Prints an information about selected command.\n\n"
     "Syntax: help [<command>]\n\n"
     "\t<command> - command we need info about\n\n",
     getHelp},

    {"create",
     "Creates a new list with specified name.\n\n"
     "Syntax: create [<modifiers>] [<name>]\n\n"
     "\t<name> - name of the list that will be created.\n\n"
     "Modifiers:\n\n"
     "\t-s, --select - selects created list as current.\n\n",
     createList},

    {"lists",
     "Prints off all available lists.\n\n",
     getLists},

    {"append",
     "Adds element to the current list.\n\n",
     addElement},

    {"show",
     "Shows the current list in table view.\n\n",
     formTable},

    {"switch",
     "Switches the current list to another with specified name.\n\n"
     "Syntax: switch [<name>]\n\n"
     "\t<name> - name of the list to switch to.\n\n",
     switchList},

    {"load",
     "Loads elements from specified file to the current list.\n\n"
     "Syntax: load [<modifiers>] [<file>]\n\n"
     "\t<file> - name of the file to load data from.\n\n"
```

```
     "Modifiers:\n\n"
     "\t-t, --text - to specify that the file type is text(default).\n"
     "\t-b, --binary - to specify that the file type is binary.\n\n",
     loadFromFile},

    {"count",
     "Counts the elements in the current list.\n\n",
     countList},

    {"delete",
     "Deletes the list with specified name.\n\n"
     "Syntax: delete [<name>]\n\n"
     "\t<name> - name of the list to be deleted.\n\n",
     deleteList},

    {"sortby",
     "Sorts list by the given parameters.\n\n"
     "Syntax: sortby [<modifiers>] <target>\n\n"
     "\t<target> - field of book which will be compared. \n"
     "\tCan be one from this:\n"
     "\tname\n\tsurname\n\ttitle\n\tyear\n\tpages\n\tprice\n\n"
     "Modifiers:\n\n"
     "\t-d, --descending - to specify that the list must be sorted in descending order.\n\n",
     sortList},

    {"filter",
     "Removes elements by specified criteria.\n\n"
     "Syntax: filter <mode> <target>\n\n"
     "\t<target> - field of book which will be checked. \n"
     "\tCan be one from this:\n"
     "\tname\n\tsurname\n\ttitle\n\tyear\n\tpages\n\tprice\n\n"
     "Modes:\n\n"
     "\tFor name, surname, title:\n"
     "\t\t--startswith <arg> - if element's target field's string starts with <arg> it will be
deleted.\n"
     "\t\t--notstartswith <arg> - if element's target field's string not starts with <arg> it
will be deleted.\n\n"
     "\tFor year, pages, price:\n"
     "\t\t--lessthan <arg> - if element's target field's value less than <arg> it will be
deleted.\n"
     "\t\t--morethan <arg> - if element's target field's value more than <arg> it will be
deleted.\n"
     "\t\t--belowaverage - if element's target field's value less than average it will be
deleted.\n"
     "\t\t--aboveaverage - if element's target field's value more than average it will be
deleted.\n\n",
     filterList},

    {"limit",
     "Limits current list to specified number of elements.\n\n"
     "Syntax: limit <number>\n\n"
     "\t<number> - number of elements in resulting list.\n\n",
     limitList},

    {"insert",
     "Inserts element at the specified position.\n\n",
     insertElement},

    {"save",
     "Saves elements from the current list to the specified file.\n\n"
     "Syntax: save [<modifiers>] [<file>]\n\n"
     "\t<file> - name of the file to save data to.\n\n"
     "Modifiers:\n\n"
     "\t-t, --text - to specify that the file type is text(default).\n"
     "\t-b, --binary - to specify that the file type is binary.\n\n",
     saveToFile}};
```

```
Command *findCommand(const char *const commandName)
{
    for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++)
    {
        if (strcmp(commands[i].commandName, commandName) == 0)
            return commands + i;
    }
    return NULL;
}

int findParam(const CLArgs *const args, const char *const param)
{
    if (!args)
        return -1;
    for (int i = 0; i < args->argc; i++)
    {
        if (strcmp(args->argv[i], param) == 0)
            return i;
    }
    return -1;
}

void addElementToList(List *list, ListItem *listItem, int pos)
{
    ListItem *iter = list->head;

    if (!iter)
        list->head = listItem;
    else if (pos == 0)
    {
        listItem->next = list->head;
        list->head = listItem;
    }
    else
    {
        int currPos = 0;
        while (iter->next != NULL && currPos + 1 < pos)
        {
            iter = iter->next;
            currPos++;
        }
        listItem->next = iter->next;
        iter->next = listItem;
    }
}

int findList(const char *const listName)
{
    for (int i = 0; i < listCount; i++)
    {
        if (strcmp((lists + i)->name, listName) == 0)
        {
            return i;
        }
    }
    return -1;
}

void deleteListRecursively(List *list)
{
    ListItem *iter = list->head;
    ListItem *next = NULL;
    while (iter)
    {
        next = iter->next;
```

```
        free(iter);
        iter = next;
    }
}

int countListElements(List *list)
{
    if (!list)
        return 0;
    int res = 0;
    for (ListItem *iter = currentList->head; iter != NULL; iter = iter->next)
        res++;
    return res;
}

void sortListUsingComparer(List *list, BookComparer comparer, int descending)
{
    int size = countListElements(list);
    if (!list || size == 0 || size == 1)
        return;

    ListItem *iter;
    Book temp;
    for (int i = 0; i < size; i++)
    {
        iter = list->head;
        while (iter->next)
        {
            if ((!descending && comparer(iter->data, iter->next->data) > 0) || (descending &&
comparer(iter->data, iter->next->data) < 0))
            {
                temp = iter->data;
                iter->data = iter->next->data;
                iter->next->data = temp;
            }
            iter = iter->next;
        }
    }
}

void filterListBy(List *list, BookPredicate predicate, ...)
{
    if (!list || !list->head)
        return;

    va_list va;
    va_start(va, predicate);
    ListItem *iter = list->head, *old, *new;
    while (list->head && predicate(list->head->data, va))
    {
        old = list->head;
        list->head = list->head->next;
        free(old);
    }

    while (iter)
    {
        while (iter->next && predicate(iter->next->data, va))
        {
            old = iter->next;
            iter->next = iter->next->next;
            free(old);
        }
        iter = iter->next;
    }
}
```

```
#pragma endregion

#pragma region CommandsRealizations

void exitProgram(const CLArgs *const args)
{

    for (int i = 0; i < listCount; i++)
    {
        deleteListRecursively(lists + i);
    }
    free(lists);

    printf("Aborting a program..\n");

    if (!args || args->argc == 0 || findParam(args, "-s") != -1 || findParam(args, "--
success") != -1)
        exit(EXIT_SUCCESS);
    else if (findParam(args, "-f") != -1 || findParam(args, "--fail") != -1)
        exit(EXIT_FAILURE);
}

void clearConsole(const CLArgs *const args)
{
    cls();
}

void getHelp(const CLArgs *const args)
{
    if (!args || args->argc == 0)
    {
        printf("There are %d commands available:\n\n", sizeof(commands) /
sizeof(commands[0]));
        for (int i = 0; i < sizeof(commands) / sizeof(commands[0]); i++)
        {
            printf("\t%s\n", commands[i].commandName);
        }
        printf("\nTo get the description of the command use \"help <command>\".\n\n");
        return;
    }
    if (args->argc == 1)
    {
        Command *command = findCommand(args->argv[0]);
        if (command)
        {
            printf(command->commandDesc);
            printf("\n");
        }
        else
        {
            printf("Unknown command \"%s\"\n\n", args->argv[0]);
            return;
        }
    }
    else
    {
        printf("Only 1 parameter needed.\n\n");
        return;
    }
}

void createList(const CLArgs *const args)
{
    if (args && args->argc >= 3)
    {
```

```
        printf("Too many parameters.\n\n");
        return;
    }
    List newList;
    newList.head = NULL;
    int selPos = findParam(args, "-s") == -1 ? findParam(args, "--select") : findParam(args,
"-s");
    int select = selPos != -1;
    if (!args || args->argc == 0 || (args->argc == 1 && select))
    {
        char buffer[MAX_LISTNAME_LEN];
        prompt("Enter list name: ", buffer);
        char *listName = trim(buffer);
        if (strlen(listName) > 0)
            strcpy(newList.name, listName);
        else
        {
            printf("List name must contain text characters!\n\n");
            return;
        }
    }
    else
    {
        strcpy(newList.name, selPos == 0 ? args->argv[1] : args->argv[0]);
    }

    if (findList(newList.name) != -1)
    {
        printf("List with the name \"%s\" already exists.\n\n");
        return;
    }

    listCount++;
    List *newListsArr = (List *)realloc(lists, sizeof(List) * listCount);
    if (!newListsArr)
    {
        printf("Error creating a list.\n");
        return;
    }
    newListsArr[listCount - 1] = newList;
    lists = newListsArr;
    printf("List \"%s\" successfully created.\n", newList.name);
    if (select || !currentList)
    {
        printf("Switching to new list.\n");
        currentListIndex = listCount - 1;
    }
    currentList = lists + currentListIndex;

    printf("\n");
}

void getLists(const CLArgs *const args)
{
    if (listCount == 0)
    {
        printf("There are no available lists. Create one by executing \"create\"
command.\n\n");
        return;
    }
    printf("There are %d available lists: \n\n", listCount);
    for (int i = 0; i < listCount; i++)
    {
        printf("\t%s\n", lists[i].name);
    }
    printf("\n");
```

```
}

void addElement(const CLArgs *const args)
{

    if (!currentList)
    {
        printf("No list to add element to. Create one by executing \"create\" command.\n\n");
        return;
    }

    if (args->argc > 0)
    {
        printf("Unknown modifiers.\n\n");
        return;
    }

    ListItem *newLI = (ListItem *)malloc(sizeof(ListItem));
    if (!newLI)
    {
        printf("Error creating list item.\n\n");
        return;
    }

    Book newBook = getBookFromConsole();

    newLI->data = newBook;
    newLI->next = NULL;

    ListItem *iter = currentList->head;

    addElementToList(currentList, newLI, countListElements(currentList));

    printf("Element successfuly added to list \"%s\"\n\n", currentList->name);
}

void formTable(const CLArgs *const args)
{

    if (!currentList)
    {
        printf("There are no list to display. Create one by executing \"create\"
command.\n\n");
        return;
    }

    ListItem *curr = currentList->head;
    if (!curr)
    {
        printf("No elements in the list \"%s\".\n\n", currentList->name);
        return;
    }

    int authorColumnWidth,
        titleColumnWidth,
        yearColumnWidth,
        pageColumnWidth,
        priceColumnWidth;

    getSizes(currentList, &authorColumnWidth, &titleColumnWidth,
             &yearColumnWidth, &pageColumnWidth, &priceColumnWidth);

    printf("\n");

    MAKE_LINE('=', '+', authorColumnWidth + 2, titleColumnWidth + 2, yearColumnWidth + 2,
pageColumnWidth + 2, priceColumnWidth + 2);
```

```
    printf("| %-*s | %-*s | %-*s | %-*s | %-*s |\n",
           authorColumnWidth, "Author",
           titleColumnWidth, "Title",
           yearColumnWidth, "Pub. year",
           pageColumnWidth, "Pages",
           priceColumnWidth, "Price");

    MAKE_LINE('=', '+', authorColumnWidth + 2, titleColumnWidth + 2, yearColumnWidth + 2,
pageColumnWidth + 2, priceColumnWidth + 2);

    char fullName[MAX_NAME_LEN + MAX_SURNAME_LEN + 1];

    while (curr)
    {
        sprintf(fullName, "%s %s", curr->data.author.name, curr->data.author.surname);
        printf("| %-*s | %-*s | %-*d | %-*.2lf |\n",
               authorColumnWidth, fullName,
               titleColumnWidth, curr->data.title,
               yearColumnWidth, curr->data.pubYear,
               pageColumnWidth, curr->data.pageCount,
               priceColumnWidth, curr->data.price);

        curr = curr->next;
    }

    MAKE_LINE('=', '+', authorColumnWidth + 2, titleColumnWidth + 2, yearColumnWidth + 2,
pageColumnWidth + 2, priceColumnWidth + 2);
    printf("\n");
}

void switchList(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("There no lists to switch between. Create one by executing \"create\"
command.\n\n");
        return;
    }
    if (args && args->argc >= 2)
    {
        printf("Too many parameters.\n\n");
        return;
    }
    char listName[MAX_LISTNAME_LEN];
    if (!args || args->argc == 0)
    {
        prompt("Enter list name to switch on: ", listName);
    }
    else if (args->argc == 1)
    {
        strcpy(listName, args->argv[0]);
    }

    char *listNameTrimmed = trim(listName);
    if (strlen(listNameTrimmed) == 0)
    {
        printf("Name of list cannot be empty.\n\n");
        return;
    }

    if (!strcmp(listNameTrimmed, currentList->name))
    {
        printf("Already at this list.\n\n");
        return;
    }
```

```
    int listIndex = -1;

    if ((listIndex = findList(listName)) != -1)
    {
        printf("Switching to list \"%s\".\n\n", listName);
        currentListIndex = listIndex;
        currentList = lists + currentListIndex;
        return;
    }

    printf("No list with name \"%s\".\n\n", listName);
}

void loadFromFile(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("No list to load data to. Create one by executing \"create\" command.\n\n");
        return;
    }
    char filename[MAX_LINE_LEN];
    int formatPointerPos = -1;
    int textFormat = 1;
    if (!args || args->argc == 0)
    {
        prompt("Enter name of file data will be loaded from: ", filename);
    }
    else if (args->argc >= 3)
    {
        printf("Too many parameters.\n\n");
        return;
    }
    else if (args->argc == 1)
    {
        if (((formatPointerPos = findParam(args, "-t")) != -1) || ((formatPointerPos =
findParam(args, "--text")) != -1))
        {
            textFormat = 1;
            prompt("Enter name of file data will be loaded from: ", filename);
        }
        else if (((formatPointerPos = findParam(args, "-b")) != -1) || ((formatPointerPos =
findParam(args, "--binary")) != -1))
        {
            textFormat = 0;
            prompt("Enter name of file data will be loaded from: ", filename);
        }
        else
        {
            strcpy(filename, args->argv[0]);
        }
    }
    else
    {

        if (((formatPointerPos = findParam(args, "-t")) != -1) || ((formatPointerPos =
findParam(args, "--text")) != -1))
        {
            textFormat = 1;
        }
        else if (((formatPointerPos = findParam(args, "-b")) != -1) || ((formatPointerPos =
findParam(args, "--binary")) != -1))
        {
            textFormat = 0;
        }
        else
```

```
        {
            printf("Unknown parameter.\n\n");
            return;
        }

        strcpy(filename, args->argv[formatPointerPos == 0 ? 1 : 0]);
    }

    char *fileNameTrimmed = trim(filename);

    if (strlen(fileNameTrimmed) == 0)
    {
        printf("The name of file cannot be empty.\n\n");
        return;
    }

    if (fexists(fileNameTrimmed))
    {

        FILE *fin = NULL;
        Book data;
        ListItem *newLI = NULL;
        int elemCount = 0;
        if (textFormat)
        {
            fin = fopen(fileNameTrimmed, "r");
            char buffer[MAX_LINE_LEN];
            while (fgets(buffer, MAX_LINE_LEN, fin) != NULL)
            {
                if (strlen(buffer) <= 5)
                    continue;
                data = strToBook(buffer);
                newLI = (ListItem *)malloc(sizeof(ListItem));
                newLI->data = data;
                newLI->next = NULL;
                addElementToList(currentList, newLI, elemCount);
                elemCount++;
            }
        }
        else
        {
            FILE *fin = fopen(fileNameTrimmed, "rb");

            while (!feof(fin))
            {
                fread(&data, sizeof(Book), 1, fin);
                if (!feof(fin))
                {
                    newLI = (ListItem *)malloc(sizeof(ListItem));
                    newLI->data = data;
                    newLI->next = NULL;
                    addElementToList(currentList, newLI, elemCount);
                    elemCount++;
                }
            }
        }

        if (elemCount)
        {
            printf("%d elements was successfully read from file \"%s\".\n\n", elemCount,
fileNameTrimmed);
        }
        else
        {
            printf("File was opened, but there are no elements to read.\n\n");
        }
```

```
        if (fin)
            fclose(fin);
    }
    else
    {
        printf("File \"%s\" not exists.\n\n", fileNameTrimmed);
    }
}

void countList(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("There are no list. Create one by executing \"create\" command.\n\n");
        return;
    }
    int res = countListElements(currentList);
    if (res == 0)
        printf("There are no elements in list.\n\n");
    else
        printf("There are %d elements in list.\n\n", res);
}

void deleteList(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("No lists to delete. Create one by executing \"create\" command.\n\n");
        return;
    }
    if (args && args->argc >= 2)
    {
        printf("Too many parameters.\n\n");
        return;
    }

    char listName[MAX_LISTNAME_LEN];
    if (!args || args->argc == 0)
    {
        prompt("Enter name of the list to be deleted: ", listName);
    }
    else if (args->argc == 1)
    {
        strcpy(listName, args->argv[0]);
    }

    char *listNameTrimmed = trim(listName);

    if (strlen(listNameTrimmed) == 0)
    {
        printf("List name cannot be empty.\n\n");
        return;
    }

    int listIndex = findList(listNameTrimmed);
    if (listIndex == -1)
    {
        printf("No list with name \"%s\".\n\n", listName);
        return;
    }

    deleteListRecursively(lists + listIndex);

    for (int i = listIndex + 1; i < listCount; i++)
    {
```

```
        lists[i - 1] = lists[i];
    }

    listCount--;

    List *newListArr = listCount ? realloc(lists, listCount) : NULL;
    lists = newListArr;

    if (!lists)
        currentListIndex = -1;
    else if (currentListIndex == listIndex)
    {
        currentListIndex = 0;
    }
    else if (currentListIndex > listIndex)
        currentListIndex--;

    currentList = lists ? lists + currentListIndex : NULL;
    printf("Successfuly deleted list with name \"%s\".\n\n", listName);
}

void sortList(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("No list to sort. Create one by executing \"create\" command.\n\n");
        return;
    }

    if (!args || args->argc == 0)
    {
        printf("Specify sort's target. To get list of available targets execute \"help
sortby\".\n\n");
        return;
    }

    if (args->argc >= 3)
    {
        printf("Too many parameters.\n\n");
        return;
    }

    char *sortTarget;
    int descending = 0;

    if (args->argc == 1)
    {
        sortTarget = args->argv[0];
    }
    else if (args->argc == 2)
    {
        int desPos = -1;
        if ((desPos = findParam(args, "-d")) != -1 || (desPos = findParam(args, "--
descending")) != -1)
        {
            descending = 1;
            sortTarget = args->argv[desPos == 0 ? 1 : 0];
        }
    }

    if (!strcmp(sortTarget, "price"))
        sortListUsingComparer(currentList, priceBookComparer, descending);
    else if (!strcmp(sortTarget, "name"))
        sortListUsingComparer(currentList, authorsNameBookComparer, descending);
    else if (!strcmp(sortTarget, "surname"))
        sortListUsingComparer(currentList, authorsSurnameBookComparer, descending);
```

```
    else if (!strcmp(sortTarget, "pages"))
        sortListUsingComparer(currentList, pageBookComparer, descending);
    else if (!strcmp(sortTarget, "year"))
        sortListUsingComparer(currentList, yearBookComparer, descending);
    else if (!strcmp(sortTarget, "title"))
        sortListUsingComparer(currentList, titleBookComaprer, descending);
    else
    {
        printf("Specify correct sort's target. To get list of available targets execute \"help
sortby\".\n\n");
        return;
    }
    printf("List sorted successfully.\n\n");
}

void filterList(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("No list to filter. Create one by executing \"create\" command.\n\n");
        return;
    }
    if (!args || args->argc > 3 || args->argc < 2)
    {
        printf("Incorrect number of parameters.\n\n");
        return;
    }

    char *target = "", *mode = "", *arg = "";
    int modePos;

    if ((modePos = findParam(args, "--lessthan")) != -1 ||
        (modePos = findParam(args, "--morethan")) != -1 ||
        (modePos = findParam(args, "--startswith")) != -1 ||
        (modePos = findParam(args, "--notstartswith")) != -1)
    {
        mode = args->argv[modePos];
        if (args->argc != 3)
        {
            printf("Parameter for \"%s\" mode needed.\n\n", mode);
            return;
        }
        if (modePos == 2)
        {
            printf("Parameter must be after mode.\n\n");
            return;
        }
        arg = args->argv[modePos == 0 ? 1 : 2];
        target = args->argv[modePos == 0 ? 2 : 0];
    }

    else if ((modePos = findParam(args, "--belowaverage")) != -1 ||
             (modePos = findParam(args, "--aboveaverage")) != -1)
    {
        mode = args->argv[modePos];
        if (args->argc != 2)
        {
            printf("No need parameter for \"%s\" mode.\n\n", mode);
            return;
        }
        target = args->argv[modePos == 0 ? 1 : 0];
    }

    else
    {
```

```
        printf("Need to specify correct filter mode. To get a list of possible modes execute
\"help filter\".\n\n");
        return;
    }

    if (!strcmp(target, "price") || !strcmp(target, "year") || !strcmp(target, "pages"))
    {

        if (!strcmp(mode, "--lessthan"))
        {
            if (isNumber(arg))
            {
                filterListBy(currentList, lessThanS, target, arg);
            }
            else
            {
                printf("Illegal argument for \"%s\" mode.\n\n", mode);
                return;
            }
        }
        else if (!strcmp(mode, "--morethan"))
        {
            if (isNumber(arg))
            {
                filterListBy(currentList, moreThanS, target, arg);
            }
            else
            {
                printf("Illegal argument for \"%s\" mode.\n\n", mode);
                return;
            }
        }
        else if (!strcmp(mode, "--belowaverage"))
        {
            double average = getListAverage(currentList, GetBooksField, target);
            printf("Average: %lf\n", average);
            filterListBy(currentList, lessThanD, target, average);
        }
        else if (!strcmp(mode, "--aboveaverage"))
        {
            double average = getListAverage(currentList, GetBooksField, target);
            printf("Average: %lf\n", average);
            filterListBy(currentList, moreThanD, target, average);
        }
        else
        {
            printf("\"%s\" mode is not compatible with \"%s\" field.\n\n", mode, target);
            return;
        }
    }

    else if (!strcmp(target, "title") || !strcmp(target, "name") || !strcmp(target,
"surname"))
    {
        if (!strcmp(mode, "--startswith"))
        {
            filterListBy(currentList, startsWithS, target, arg);
        }
        else if (!strcmp(mode, "--notstartswith"))
        {
            filterListBy(currentList, notStartsWithS, target, arg);
        }
        else
        {
            printf("\"%s\" mode is not compatible with \"%s\" field.\n\n", mode, target);
            return;
```

```
        }
    }

    printf("Filter was successful.\n\n");
}

void limitList(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("No list to limit. Create one by executing \"create\" command.\n\n");
        return;
    }
    if (!args || args->argc != 1)
    {
        printf("Illegal number of parameters.\n\n");
        return;
    }

    int lim = atoi(args->argv[0]);
    int size = countListElements(currentList);

    if (lim <= 0)
    {
        printf("Illegal parameter for limit.\n\n");
        return;
    }

    if (size <= lim)
    {
        printf("There are already less elements in list.\n\n");
        return;
    }

    ListItem *iter = currentList->head;
    for (int i = 0; i < lim - 1; i++)
    {
        iter = iter->next;
    }

    ListItem *old = iter->next, *temp = iter->next->next;
    iter->next = NULL;
    while (old)
    {
        temp = old->next;
        free(old);
        old = temp;
    }

    printf("List limited successfuly.\n\n");
}

void insertElement(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("No list to insert element to. Create one by executing \"create\"
command.\n\n");
        return;
    }
    if (args && args->argc != 0)
    {
        printf("No parameters needed.\n\n");
        return;
    }
```

```
    ListItem *newLI = (ListItem *)malloc(sizeof(ListItem));

    if (!newLI)
    {
        printf("Error creating new list item.\n\n");
        return;
    }

    Book newBook = getBookFromConsole();

    int pos = 0;
    printf("Enter position of this item in list: ");
    scanf("%d", &pos);

    if (pos <= 0)
    {
        printf("Illegal position.\n\n");
        return;
    }

    newLI->data = newBook;
    newLI->next = NULL;

    addElementToList(currentList, newLI, pos - 1);

    printf("Element was added successfully.\n\n");
}

void saveToFile(const CLArgs *const args)
{
    if (!currentList)
    {
        printf("No list to save data from. Create one by executing \"create\" command.\n\n");
        return;
    }
    char filename[MAX_LINE_LEN];
    int formatPointerPos = -1;
    int textFormat = 1;
    if (!args || args->argc == 0)
    {
        prompt("Enter name of file data will be saved to: ", filename);
    }
    else if (args->argc >= 3)
    {
        printf("Too many parameters.\n\n");
        return;
    }
    else if (args->argc == 1)
    {
        if (((formatPointerPos = findParam(args, "-t")) != -1) || ((formatPointerPos =
findParam(args, "--text")) != -1))
        {
            textFormat = 1;
            prompt("Enter name of file data will be saved to: ", filename);
        }
        else if (((formatPointerPos = findParam(args, "-b")) != -1) || ((formatPointerPos =
findParam(args, "--binary")) != -1))
        {
            textFormat = 0;
            prompt("Enter name of file data will be saved to: ", filename);
        }
        else
        {
            strcpy(filename, args->argv[0]);
        }
    }
```

```
    else
    {

        if (((formatPointerPos = findParam(args, "-t")) != -1) || ((formatPointerPos =
findParam(args, "--text")) != -1))
        {
            textFormat = 1;
        }
        else if (((formatPointerPos = findParam(args, "-b")) != -1) || ((formatPointerPos =
findParam(args, "--binary")) != -1))
        {
            textFormat = 0;
        }
        else
        {
            printf("Unknown parameter.\n\n");
            return;
        }

        strcpy(filename, args->argv[formatPointerPos == 0 ? 1 : 0]);
    }

    char *fileNameTrimmed = trim(filename);

    if (strlen(fileNameTrimmed) == 0)
    {
        printf("File name cannot be empty.\n\n");
        return;
    }

    if (fexists(fileNameTrimmed))
    {
        printf("The file \"%s\" already exists.\n", fileNameTrimmed);
        char choice[3];
        prompt("Do you want to overwrite it?(Y|N) ", choice);
        if (!(tolower(*choice) == 'y'))
        {
            printf("\n\n");
            return;
        }
    }

    FILE *fout = NULL;

    if (textFormat)
    {
        fout = fopen(fileNameTrimmed, "w");

        if (!fout)
        {
            printf("Error accessing the file.\n\n");
            return;
        }

        ListItem *iter = currentList->head;
        Book currentData;
        while (iter)
        {
            currentData = iter->data;
            fprintf(fout, "%s %s,%s,%d,%d,%.2lf\n",
                    currentData.author.name,
                    currentData.author.surname,
                    currentData.title,
                    currentData.pubYear,
                    currentData.pageCount,
                    currentData.price);
```

```
            iter = iter->next;
        }
    }
    else
    {
        fout = fopen(fileNameTrimmed, "wb");

        ListItem *iter = currentList->head;
        Book currentData;
        while (iter)
        {
            currentData = iter->data;
            fwrite(&currentData, sizeof(Book), 1, fout);
            iter = iter->next;
        }
    }

    if (fout)
        fclose(fout);

    printf("Saving was successful.\n\n");
}

#pragma endregion

#pragma region Console

CLArgs *parseCommandLine(char *str, char **command)
{

    CLArgs *result = (CLArgs *)malloc(sizeof(CLArgs));
    result->argc = 0;

    str = trim(str);

    char *commandArgSepPos = strchr(str, ' ');

    *command = str;
    //if no args(no space after command)
    if (!commandArgSepPos)
    {
        return result;
    }

    *commandArgSepPos = 0;

    int inQuote = 0;
    char *lexemStart = ltrim(commandArgSepPos + 1);
    char *curr = lexemStart;
    while (*curr != 0)
    {

        if (!inQuote && *curr == '"' && strchr(curr + 1, '"'))
        {
            inQuote = 1;
        }
        else if (*curr == '"' && inQuote)
        {
            inQuote = 0;
        }

        if (!inQuote && isspace(*curr))
        {
            *curr = 0;
            result->argv[result->argc++] = qtrim(trim(lexemStart));
```

```
            curr = ltrim(curr + 1);
            lexemStart = curr;
        }
        else
        {
            curr++;
            if (*curr == 0)
            {
                result->argv[result->argc++] = qtrim(trim(lexemStart));
            }
        }
    }

    return result;
}

void executeLine(char *line)
{
    char buffer[20][MAX_LINE_LEN];
    char *part = strtok(line, ";");
    int commandCount = 0;
    while (part)
    {
        if (strlen(part) > 1)
            strcpy(buffer[commandCount++], part);
        part = strtok(NULL, ";");
    }

    for (int i = 0; i < commandCount; i++)
    {
        char *commandName;
        CLArgs *args = parseCommandLine(buffer[i], &commandName);

        Command *command;

        if (command = findCommand(commandName))
        {
            command->commandAction(args);
            fflush(stdin);
        }
        else
        {
            printf("Unknown command \"%s\"\nType \"help\" to get list of all available
commands.\n\n", commandName);
        }
    }
}

void executeFile(char *fileName)
{
    if (!rightFileFormat(fileName))
    {
        printf("Use only .lsexe files.\n\n");
        return;
    }
    FILE *fin = fopen(fileName, "r");
    if (!fin)
    {
        printf("Error opening file \"%s\".", fileName);
        return;
    }
    char buffer[MAX_LINE_LEN];
    char consoleText[MAX_LISTNAME_LEN + 2];
    while (fgets(buffer, MAX_LINE_LEN, fin))
    {
        sprintf(consoleText, "%s> ", currentList ? currentList->name : "(no lists)");
```

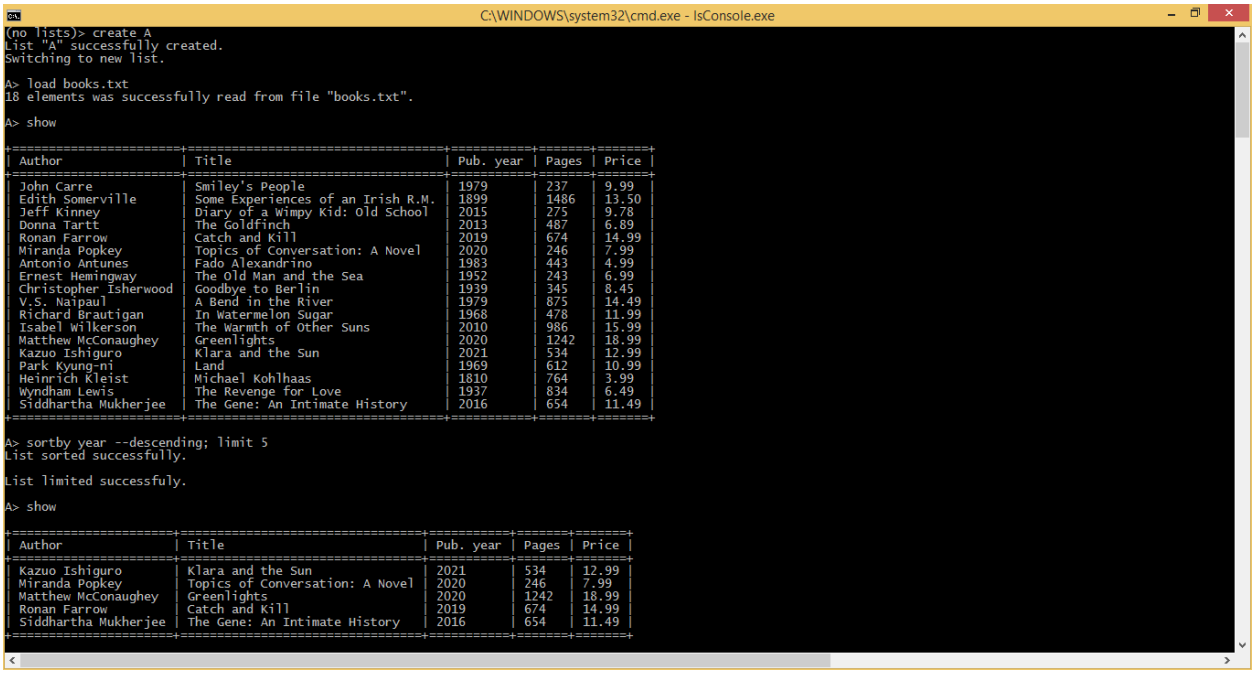
```
        executeLine(buffer);
    }
}

void launchConsole()
{
    char buffer[MAX_LINE_LEN];
    while (1)
    {
        char consoleText[MAX_LISTNAME_LEN + 2];
        sprintf(consoleText, "%s> ", currentList ? currentList->name : "(no lists)");
        prompt(consoleText, buffer);

        executeLine(buffer);
    }
}

#pragma endregion
```

(lab10.c):

```
#include "lab10_lib.h"

int main(int argc, char *argv[])
{
    cls();

    if (argc == 2)
    {
        executeFile(argv[1]);
    }

    launchConsole();
    return 0;
}
```

Протокол роботи:

Програма при запуску очищає консоль і запускає на виконання власну оболонку для виконання команд по опрацюванню списків. Там ми можемо вводити певні команди, що будуть виконанні інтерпретатором:



Також програма має опцію обробки скріптових файлів, що містять задані наперед команди. В одному з таких файлів я і реалізував індивідуальне завдання.

Скріпт (lab10_script.lsexe):

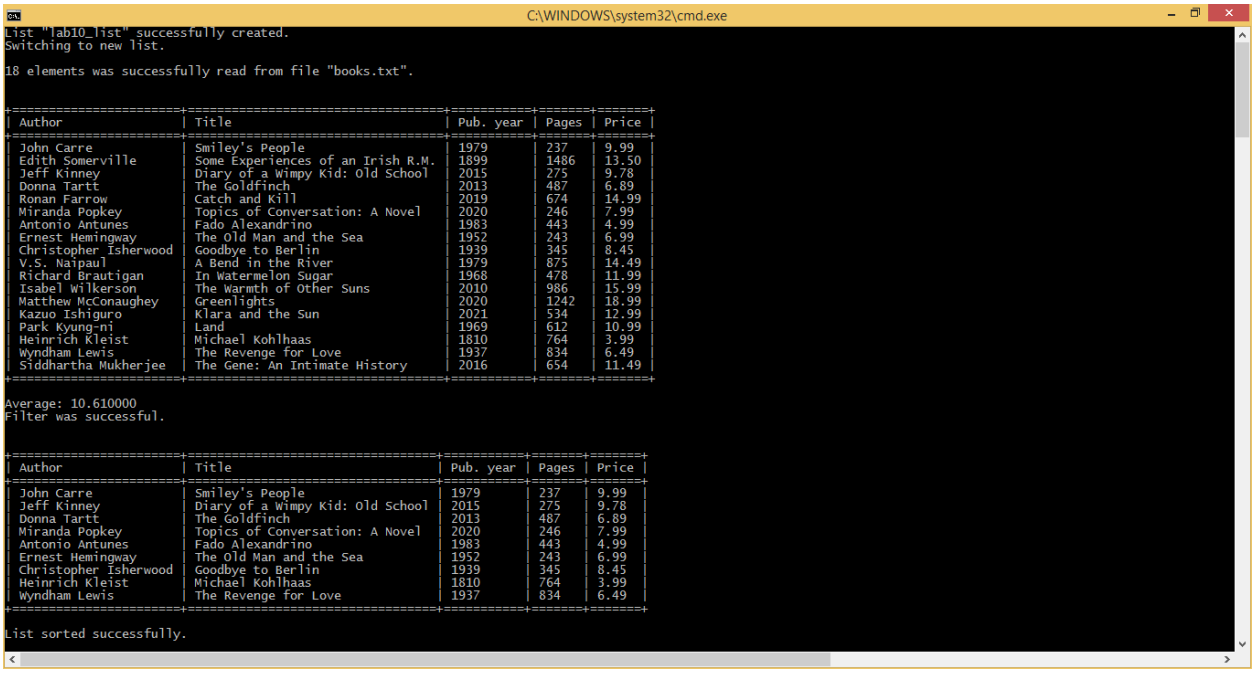
```
create lab10_list;
load books.txt; show;
filter price --aboveaverage; show;
sortby surname; show;
filter price --lessthan 5; show;
save result.txt
exit
```

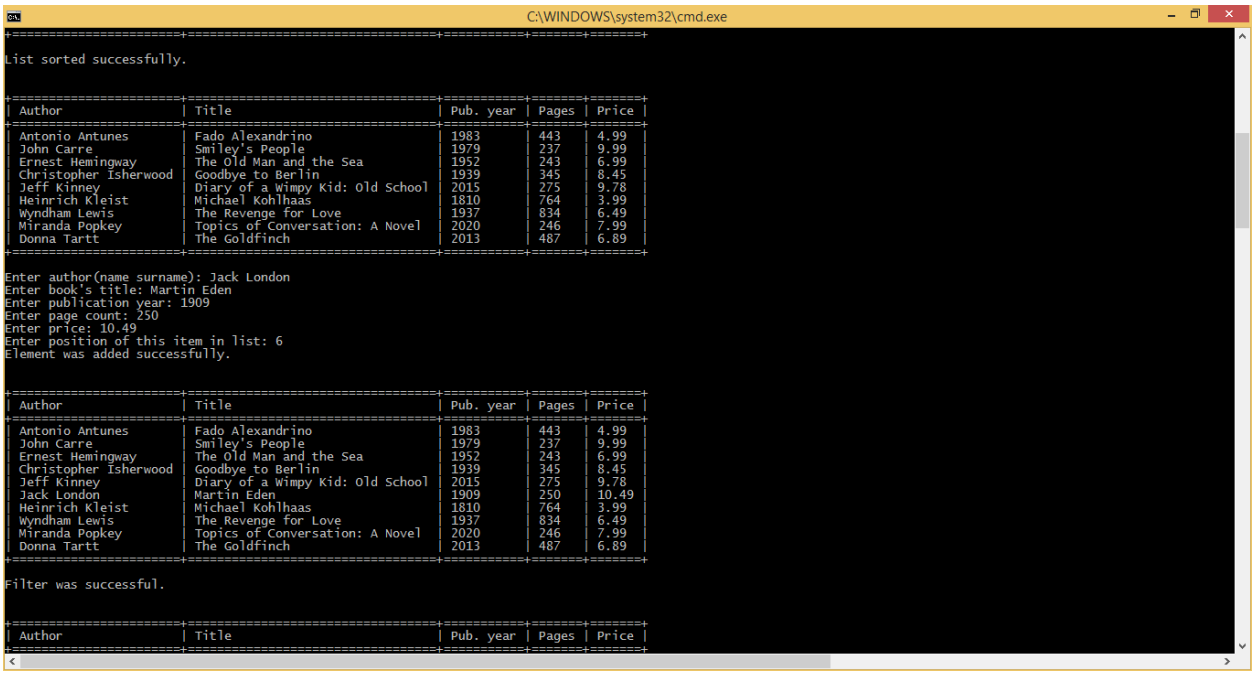
Вхідний файл(books.txt):

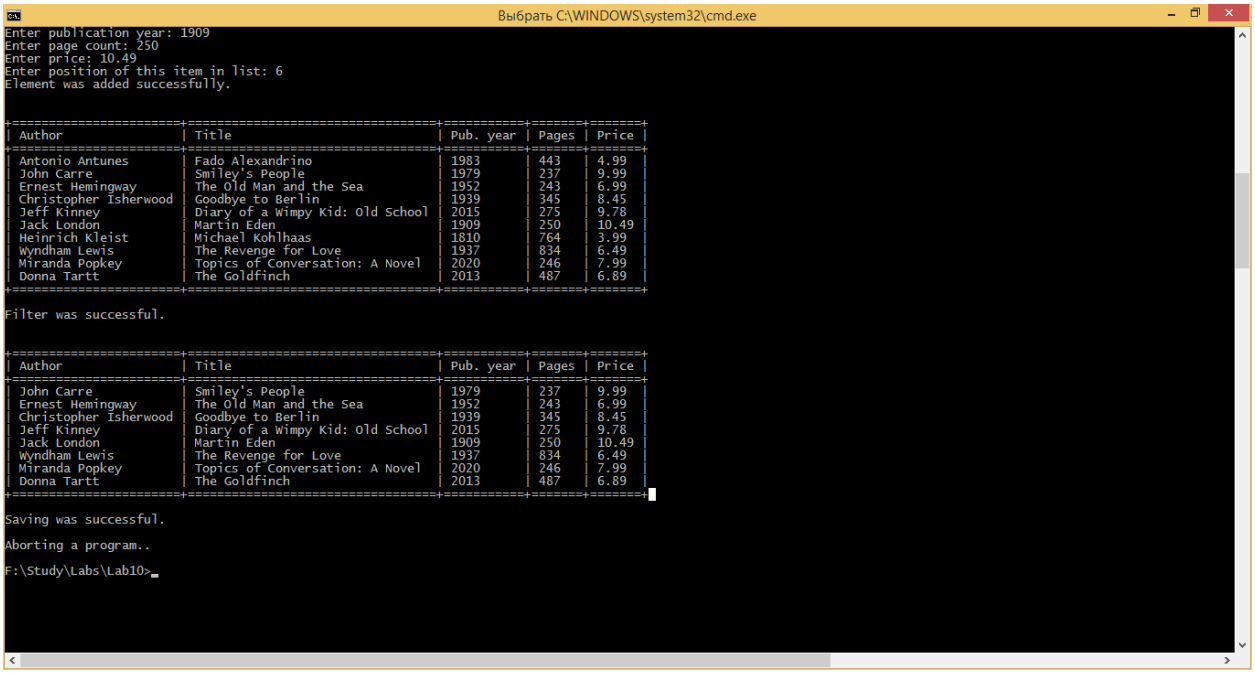
```
John Carre,Smiley's People,1979,237,9.99
Edith Somerville,Some Experiences of an Irish R.M.,1899,1486,13.5
Jeff Kinney,Diary of a Wimpy Kid: Old School,2015,275,9.78
Donna Tartt,The Goldfinch,2013,487,6.89
Ronan Farrow,Catch and Kill,2019,674,14.99
Miranda Popkey,Topics of Conversation: A Novel,2020,246,7.99
Antonio Antunes,Fado Alexandrino,1983,443,4.99
Ernest Hemingway,The Old Man and the Sea,1952,243,6.99
Christopher Isherwood,Goodbye to Berlin,1939,345,8.45
V.S. Naipaul,A Bend in the River,1979,875,14.49
```

Richard Brautigan,In Watermelon Sugar,1968,478,11.99
Isabel Wilkerson,The Warmth of Other Suns,2010,986,15.99
Matthew McConaughey,Greenlights,2020,1242,18.99
Kazuo Ishiguro,Klara and the Sun,2021,534,12.99
Park Kyung-ni,Land,1969,612,10.99
Heinrich Kleist,Michael Kohlhaas,1810,764,3.99
Wyndham Lewis,The Revenge for Love,1937,834,6.49
Siddhartha Mukherjee,The Gene: An Intimate History,2016,654,11.49

Результат виконання:







Як бачимо, програма успішно відпрацювала і отриманий результат записала у вихідний файл.

Вихідний файл(result.txt):

John Carre,Smiley's People,1979,237,9.99
Ernest Hemingway,The Old Man and the Sea,1952,243,6.99
Christopher Isherwood,Goodbye to Berlin,1939,345,8.45
Jeff Kinney,Diary of a Wimpy Kid: Old School,2015,275,9.78
Jack London,Martin Eden,1909,250,10.49
Wyndham Lewis,The Revenge for Love,1937,834,6.49
Miranda Popkey,Topics of Conversation: A Novel,2020,246,7.99
Donna Tartt,The Goldfinch,2013,487,6.89

Висновок:

Написання цієї лабараторної роботи дозволило нам познайомитись з використанням динамічних струкур, а в цьому конкретному випадку – однозв’язними списками. Можливо, код вийшов трохи громіздким, але додавання додаткового функціоналу допомогло нам повторити ті теми, які були освітлені у попередніх 9 ЛР.