МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №2

по дисциплине «Алгоритмы и структуры данных»

Тема: Иерархические списки

Студент гр. 9381 _____ Птичкин С. А.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

Цель работы.

Научиться использовать иерархические списки для решения задач. Изучить принципы работы с ними, особенности их использования.

Задание.

Вариант 13.

Вычислить глубину (число уровней вложения) иерархического списка как максимальное число одновременно открытых левых скобок в сокращённой скобочной записи списка; принять, что глубина пустого списка и глубина атомарного S-выражения равны нулю; например, глубина списка (a (b ( ) c) d) равна двум.

Выполнение работы.

Для применения некоторых функций были подключены заголовочные файлы iostream, cstdio, string.h, fstream.

Описание алгоритма.

Для вычисления глубины списка используется рекурсивный алгоритм, который работает с сформированным списком. На вход рекурсивной функции подаётся голова списка. Сначала проверяется, является ли список атомом или же он пустой, в этом случае возвращается глубина равная нулю, по условию задачи. Иначе происходит рекурсивный вызов этой же функции для головы следующего списка. Таким образом алгоритм сначала проходит головы всех списков, идущих подряд, а затем, встретив пустой список или атом, начинает проверять хвост текущего списка. В конечном итоге рекурсивные функции последовательно завершаются, возвращая наибольшую глубину из головы и

хвоста. При этом значение из головы увеличивается на 1, так как это и отражает глубину списка.

Пользовательские типы данных.

1) Структура two_ptr.

   Предназначена для хранения указателей на голову и хвост иерархического списка. Поле hd - указывает на голову, tl - на хвост.

2) Структура s_expr.

   Отображает элемент иерархического списка. Содержит поле tag типа bool, значение true показывает, что элемент - атом, значение false, что элемент - пара. Объединение node содержит поля atom типа char и pair типа two_ptr. Был создан синоним LIST, с помощью директивы typedef.

3) Структура list_and_rez.

   Предназначена для хранения списка (поле list) и соответствующего ему результата применённого алгоритма(поле rez). Был создан синоним LIST_AND_REZ, с помощью директивы typedef.

Структура программы.

Программа разбита на 3 файла: list.h, хранящий определения структур и объявления функций для работы со списками. list.cpp, хранящий реализацию функций для работы со списками. main.cpp, в которой реализован пользовательский интерфейс и алгоритм обработки данных, функции консольного и файлового ввода/вывода.

Функции для работы со списками.

1) **Функция head. LIST* head(const LIST* s)**

   Данная функция принимает на вход указатель на список. Проверяет его на пустоту и на его атомарность, если он не пуст и не атом, возвращает указатель hd поля pair объединения node, иначе выдаёт ошибку.

2) **Функция tail. LIST* tail(const LIST* s)**

   Данная функция принимает на вход указатель на список. Проверяет его на пустоту и на его атомарность, если он не пуст и не атом, возвращает указатель tl поля pair объединения node, иначе выдаёт ошибку.

3) **Функция cons. LIST* cons(LIST* h, LIST* t)**

   Функция предназначена для создания пары. На вход подаются два указателя на список. Сначала список t проверяется на атом, так как хвост не может им являться. Если всё в порядке, выделяется память под объект списка, его поле tag принимает значение false, а поля hd и tl принимают значения h и t соответственно. Функция возвращает указатель на созданную пару.

4) **Функция make_atom. LIST* make_atom(const char x)**

   Данная функция предназначена для создания атомарного списка. На вход подаётся символ x. Динамически выделяется память под объект списка, его поле tag принимает значение true, в поле node.atom записывается значение x. Возвращается указатель на созданный атомарный список.

5) **Функция is_Atom. bool is_Atom(const LIST* s)**

   На вход подаётся указатель на список. Функция возвращает значение true, если список является атомом, иначе - false.

6) **Функция is_Null. bool is_Null(const LIST* s)**

На вход подаётся указатель на список. Функция возвращает значение true, если список пустой, иначе - false.

7) **Функция destroy. void destroy(LIST* s)**

На вход подаётся указатель на список. Функция предназначена для удаления элемента списка. Если список не пуст и не является атомом, то идёт рекурсивный вызов этой же функции для удаления списков, на которые указывают поля hd и tl. Функция ничего не возвращает.

8) **Функция copy_list. LIST* copy_list(const LIST* x)**

На вход подаётся указатель на список. Функция предназначена для корректного копирования элементов списка. Если список пуст, возвращается NULL, если список - атом, вызывается функция make_atom() для значения в x->node.atom, созданный атом возвращается. Иначе вызывается функция cons, для копий списков, на которые указывают поля hd и tl. Созданную пару функция возвращает.

9) **Функция write_list. void write_list(const LIST* x, ostream* stream)**

На вход функции подаётся указатель на список и указатель на поток вывода(указывает на cout при консольном выводе, и на объект fstream при файловом). Это основная функция для вывода списка в поток вывода. Если переданный список пуст, то печатается “()”. Если это атом, печатается значение его поля node.atom. Иначе вызывается функция write_seq с теми же входными данными. Вызов происходит после вывода “(”, а после её отработки выводится “)”. Функция ничего не возвращает.

10) **Функция write_seq. void write_seq(const LIST* x, ostream* stream)**

Эта функция предназначена для печати выражения внутри скобок сокращённой записи списка. На вход функции подаётся указатель на

список и указатель на поток вывода(указывает на cout при консольном выводе, и на объект fstream при файловом). Если список не пуст вызываются функции write_list и эта же функция с списком из поля hd и tl соответственно. Поток остаётся тем же. Функция ничего не возвращает.

11) **Функция read_list. void read_list(LIST*&, int* file_end_flag, istream* stream)**

Функция принимает на вход указатель на список, который передаётся по ссылке, указатель на флаг, отслеживающий достижение конца файла(имеет значение nullptr при считывании из консоли) и указатель на поток ввода, указывающий на cin при считывании с консоли, указывающий на объект типа fstream при считывании из файла. Функция считывает первый символ строки списка, пропуская пробелы. При считывании перехода на новую строку функция завершается, в итоге получается пустой список. Также проверяется достижение конца файла. Иначе вызывается функция read_s_expr, куда подаются считанный символ, указатель y и указатель на поток ввода. После считывания списка таким образом функция дочитывает строку до символа переноса строки, так как в строке подразумевается наличие 1 списка. Функция ничего не возвращает.

12) **Функция read_s_expr. void read_s_expr(char prev, LIST*& y, istream* stream)**

Функция принимает на вход ранее считанный символ, указатель на список, который передаётся по ссылке, и указатель на поток ввода, указывающий на cin при считывании с консоли, указывающий на объект типа fstream при считывании из файла. Если ранее считанный символ был “)”, это означает ошибку введения сокращённой формы записи. В консоль

выводится сообщение об ошибке и программа полностью завершается. Если предыдущим символом была не “(”, то вызывается функция make_atom для этого символа, адрес атомарного списка записывается в y. Иначе происходит считывание выражения внутри скобок, посредством вызова функции read_seq для указателя y и потока stream. Функция ничего не возвращает.

13) Функция read_seq. void read_seq(LIST*& y, istream* stream)

Функция принимает на вход указатель на список, который передаётся по ссылке и указатель на поток ввода, указывающий на cin при считывании с консоли, указывающий на объект типа fstream при считывании из файла. Сначала объявляются два указателя на элемент списка p1 и p2, затем идёт проверка конца файла. Если всё в порядке идёт считывание символа из потока stream, все пробелы пропускаются. Если считывается “)”, значит мы дошли до конца выражения в скобках, указатель y принимает значение nullptr. Иначе идёт рекурсивный вызов функций read_s_expr для данного символа и указателя p1 и вызов read_seq, для указателя p2. Поток остаётся тем же. После этого вызывается функция создания пары cons, на вход идут p1 и p2. Адрес созданной пары записывается в y. Функция ничего не возвращает.

Основные функции.

1) Функция main. int main()

Функция не принимает никаких параметров. Данная функция предназначена для стартового диалога с пользователем. Здесь идёт выбор ввода данных, либо из консоли, либо из файла. За корректность введённых данных отвечает функция int_num. Ввод команды 1 вызывает функцию консольного

ввода, команда 2 - файлового. Возвращаемые значения этих функций определяют, будет ли цикл продолжаться с возможностью ввести новые данные, либо программа завершится.

2) Функция file_input. int file_input()

Функция не принимает никаких параметров. В начале объявляются все необходимые для работы переменные, выделяется память под имя файла. Затем считыватся имя файла, и файл открывается при корректном имени. Затем объявляется массив структур LIST_AND_REZ. В поле list функция read_list, в которую был передан указатель на список по ссылке, адрес флага конца файла и адрес потока нашего файла, записывает адреса созданных списков, считанных из файла. Затем файл закрывается и вызывается data_analis, функция анализа данных массива data_mass. Функция возвращает то же значение, что и data_analis.

3) Функция console_input. int console_input()

Функция не принимает никаких параметров. В начале у пользователя запрашивается количество строк для ввода с помощью функции input_num. Затем выделяется память под массив структур LIST_AND_REZ с таким же числом элементов. В поле list функция read_list, в которую был передан указатель на список по ссылке, нулевой указатель конца файла и адрес потока cin, записывает адреса созданных списков, считанных из консоли. Функция возвращает то же значение, что и data_analis.

4) Функция data_analis. int data_analis(LIST_AND_REZ* data_mass, int count_of_list)

Функция принимает на вход массив структур LIST_AND_REZ и количество элементов массива. В начале работы в консоль выводится строка символов “-” для разграничения области ввода и обработки данных. Выводятся

исходные данные. Затем в цикле for для каждой строки массива структур LIST_AND_REZ вызывается рекурсивная функция проверки rec_func_depth, куда передаётся указатель на список list и стартовая глубина - 0. Возвращаемое значение функции записывается в поле rez каждой структуры массива. Затем выводится итоговая глубина каждого списка. После вывода результатов пользователь выбирает дальнейшие действия. Выбор команды осуществляется в бесконечном цикле, выход из которого, только при вводе корректной команде. Команда считывается функцией input_num. Значения команд: 1 - Сохранить результаты в файл и продолжить, 2 - Сохранить данные в файл и выйти, 3 - Продолжить без сохранения, 4 - Выйти без сохранения. В первых двух случаях вызываются функции data_save, затем функция очистки памяти clear_memory , в остальных двух случаях просто очистка памяти. Возвращаемое значение функции: 1 - для продолжения работы, 0 - для завершения.

5) Функция rec_func_depth. int rec_func_depth(LIST* list, int mid_depth)

Данная рекурсивная функция предназначена для нахождения глубины списка. На вход функции подаётся указатель на список и текущий уровень глубины, предназначенный для промежуточных результатов. В начале проверяются 2 условия, если список пустой или является атомом, функция возвращает 0. Также, при атомарном списке выводится текущий уровень глубины. Далее считается два значения: глубина списка в поле node.hd и node.tl. Они считаются посредством рекурсивного вызова этой функции для этих двух указателей и записываются в поля hd_depth и tl_depth соответственно. Причём глубина увеличивается 1 только при переходе из головы, при переходе из хвоста остаётся той же. Далее вычисляется максимальная из глубин hd_depth и tl_depth. Наибольшая возвращается функцией.

6) Функция clear_memory. void clear_memory(LIST_AND_REZ* data_mass, int count_of_list)

На вход функции передаётся массив структур LIST_AND_REZ и количество элементов массива. Вызывается рекурсивная функция уничтожения списка destroy. В конце очищается память, выделенная под массив data_mass.

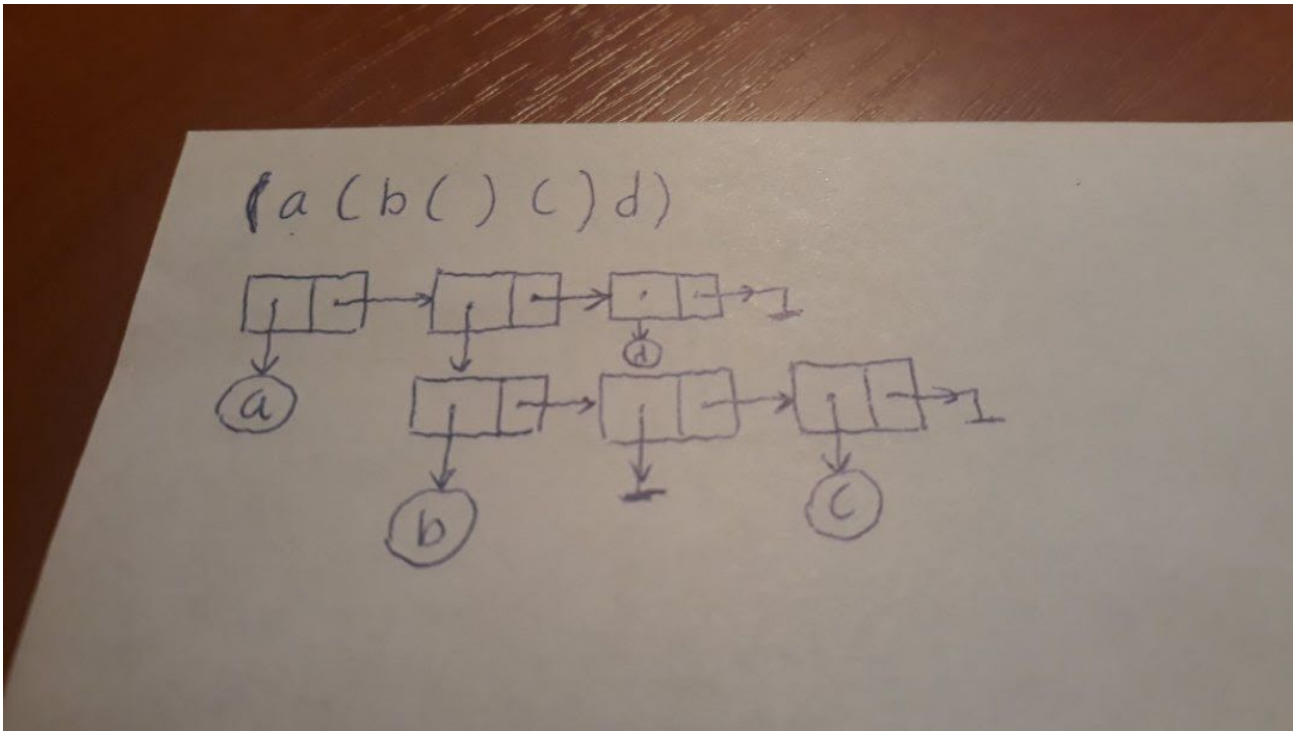
7) Функция data_save. void data_save(LIST_AND_REZ* data_mass, int count_of_list)

На вход функции передаётся массив структур LIST_AND_REZ и количество элементов массива. Данная функция предназначена для сохранения данных в файл. В начале выделяется память под имя файла, затем оно считывается из потока ввода. Открывается файл для записи, либо же он создаётся. При открытии выбирается режим записи с дозаписью, для сохранения предыдущих данных. С помощью функции write_list в открытый файл записываются сокращённые записи списка, а после каждой строки указывается результат проверки глубины списка. После записи файл закрывается. Функция ничего не возвращает.

8) Функция input_num. int input_num(char* message)

Функция принимает на вход сообщение, выводимое пользователю. Функция предназначена для корректного считывания числа из потока ввода. На вход принимается адрес строки с сообщением пользователю, что ему делать. Объявляется переменная для записи числа и выделяется буфер на 10 символов. Затем из cin считывается 10 символов в буфер. Далее в цикле из данного буфера считывается число функцией sscanf. Пока функция не вернёт 1 - количество верно считанных аргументов, ввод не прекратится. Когда наконец число считается, оно возвращается функцией. Память, выделенная под буфер очищается.

Графическая схема иерархического списка.



Тестирование.

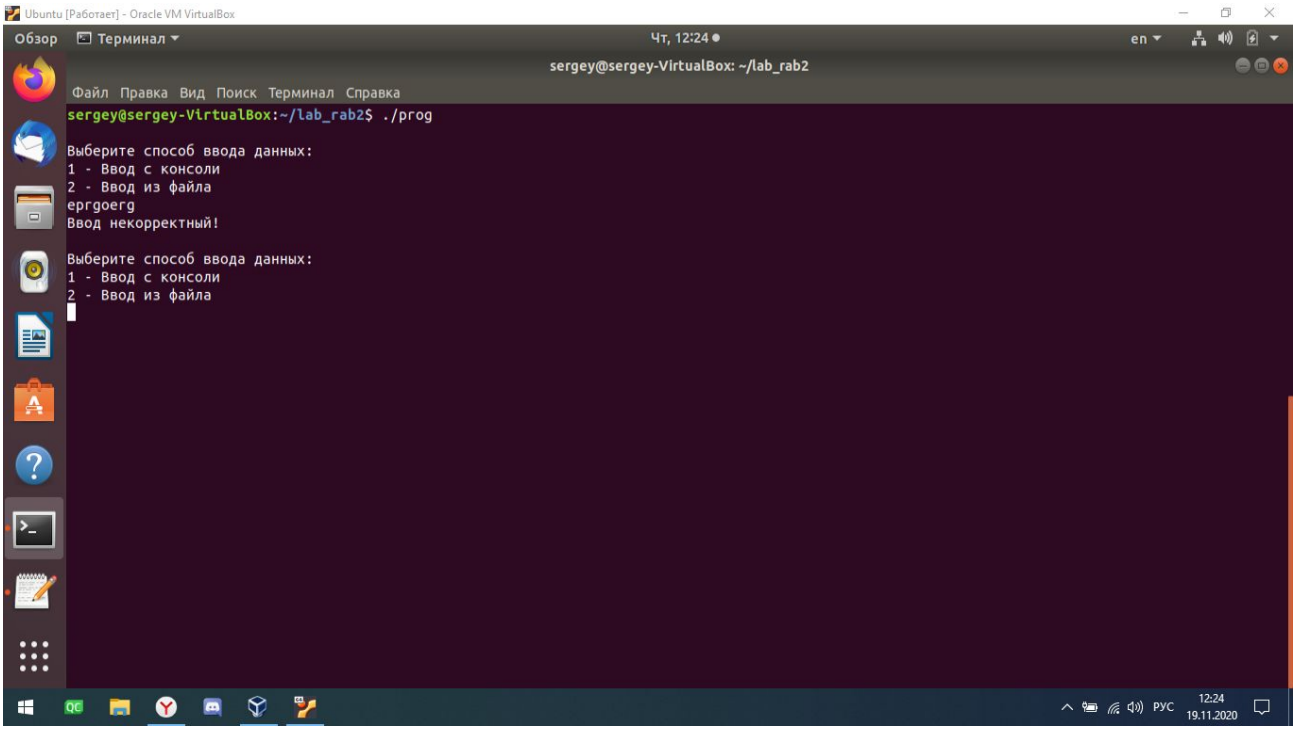
Результаты теста представлены в таблице 1.

Таблица 1- Результаты тестирования

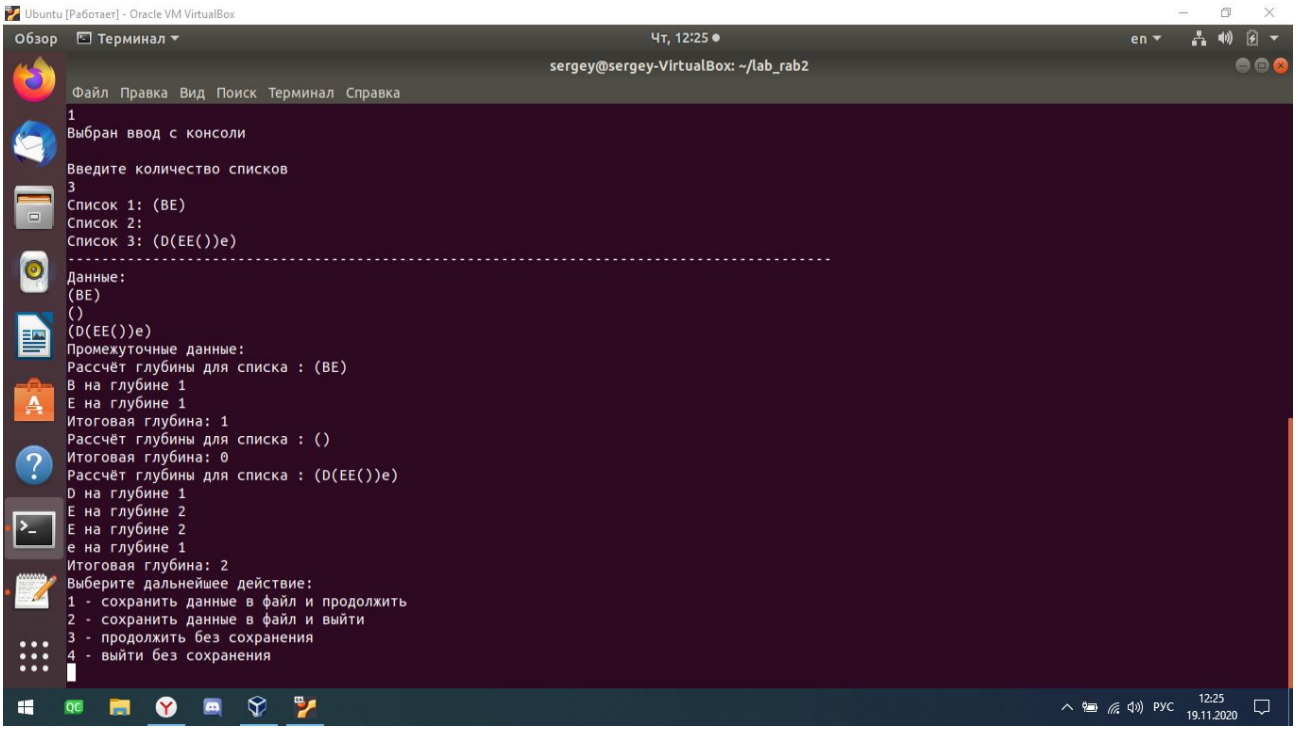
№	Входные данные	Выходные данные
1.	(DA)	(DA) Глубина: 1
2.	(dasd)as)	(dasd) Глубина: 1
3.	(S(AD)D)	(S(AD)D) Глубина: 2
4.	(G()F()F)	(G()F()F) Глубина: 1
5.	((d((c((vd))))))	((d((c((vd)))))) Глубина: 6
6.	((((()))))	((((())))) Глубина: 4
7.		()
8.	(()()())	(()()()) Глубина: 1
9	(b v (B D)( ) ff)	(bv(BD)()ff) Глубина: 2

Обработка исключительных ситуаций.

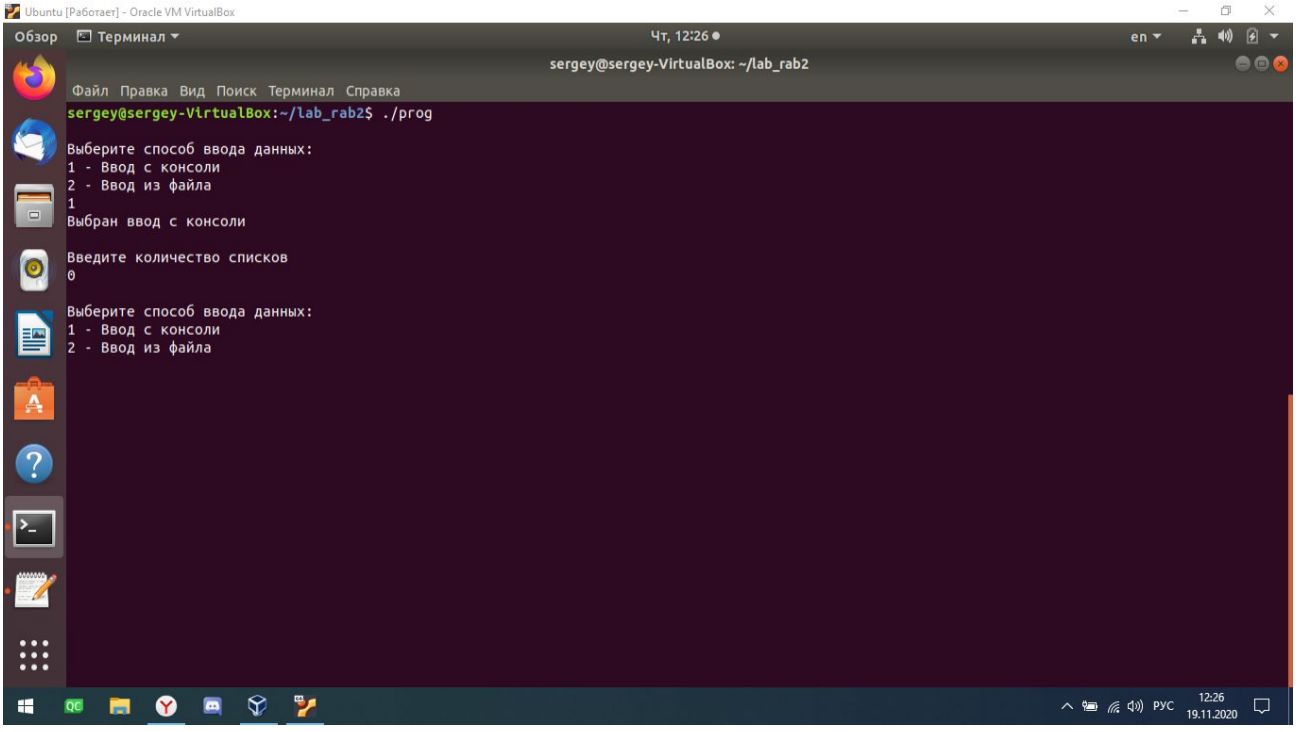
1) **Некорректные действия**



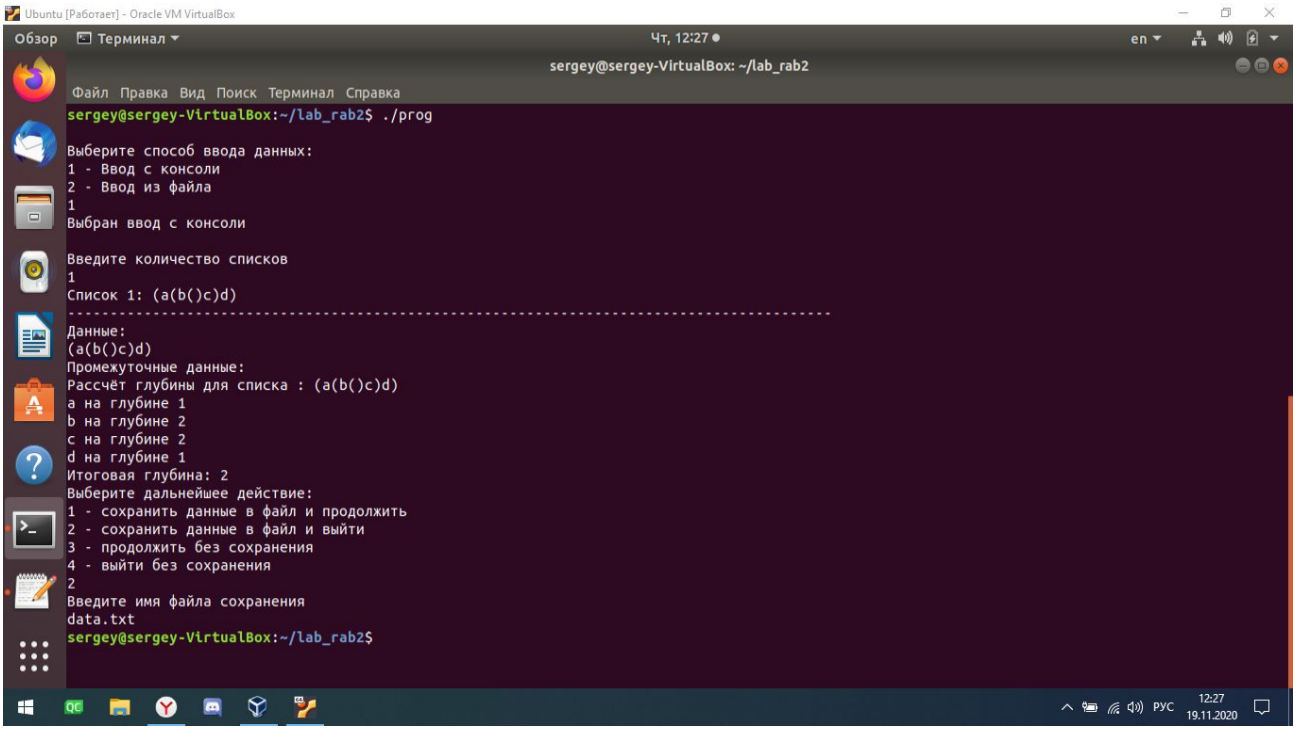
2) **Ввод с консоли**



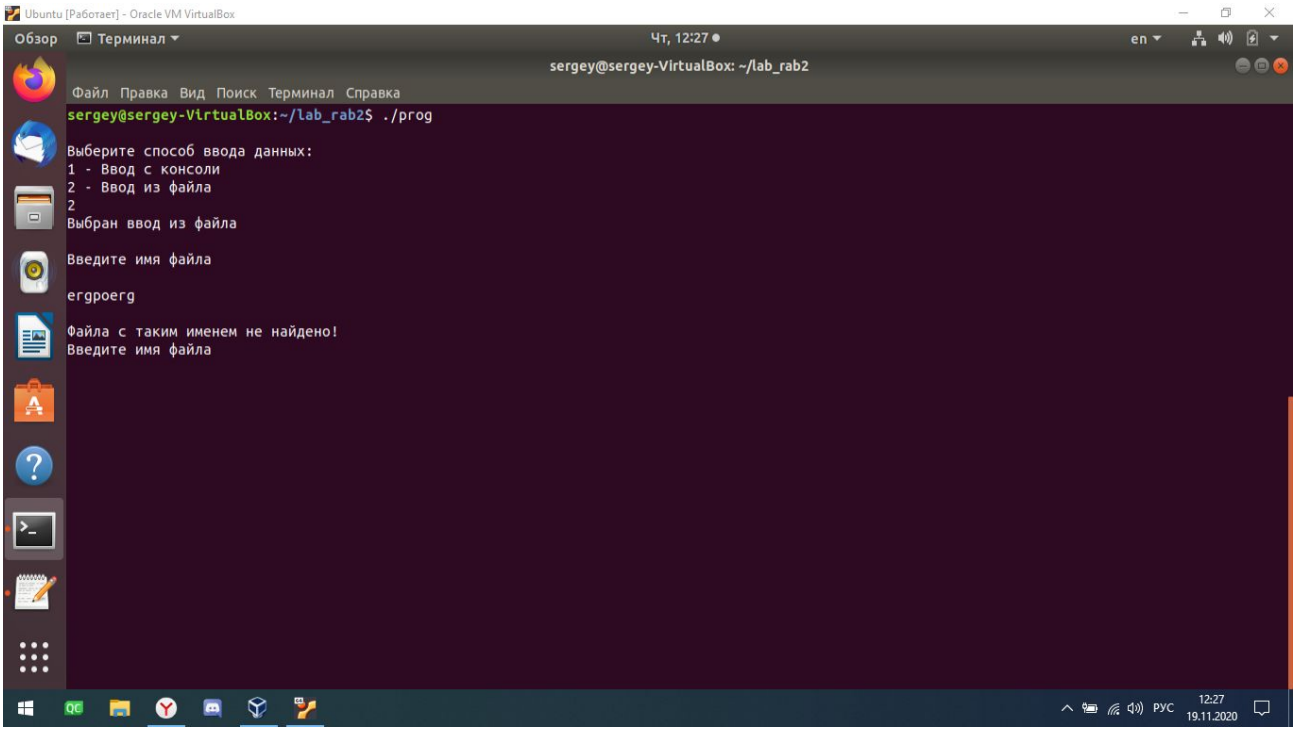
3) Некорректное количество строк



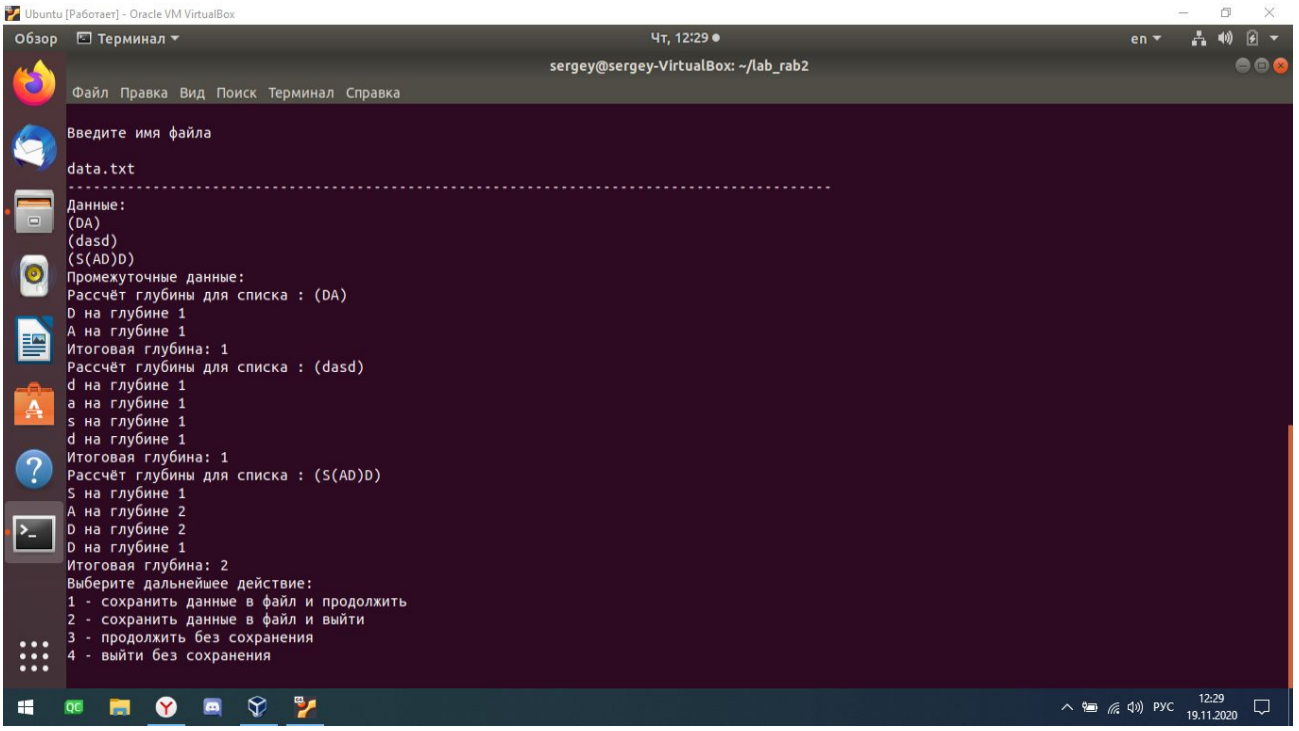
4) Сохранение в файл



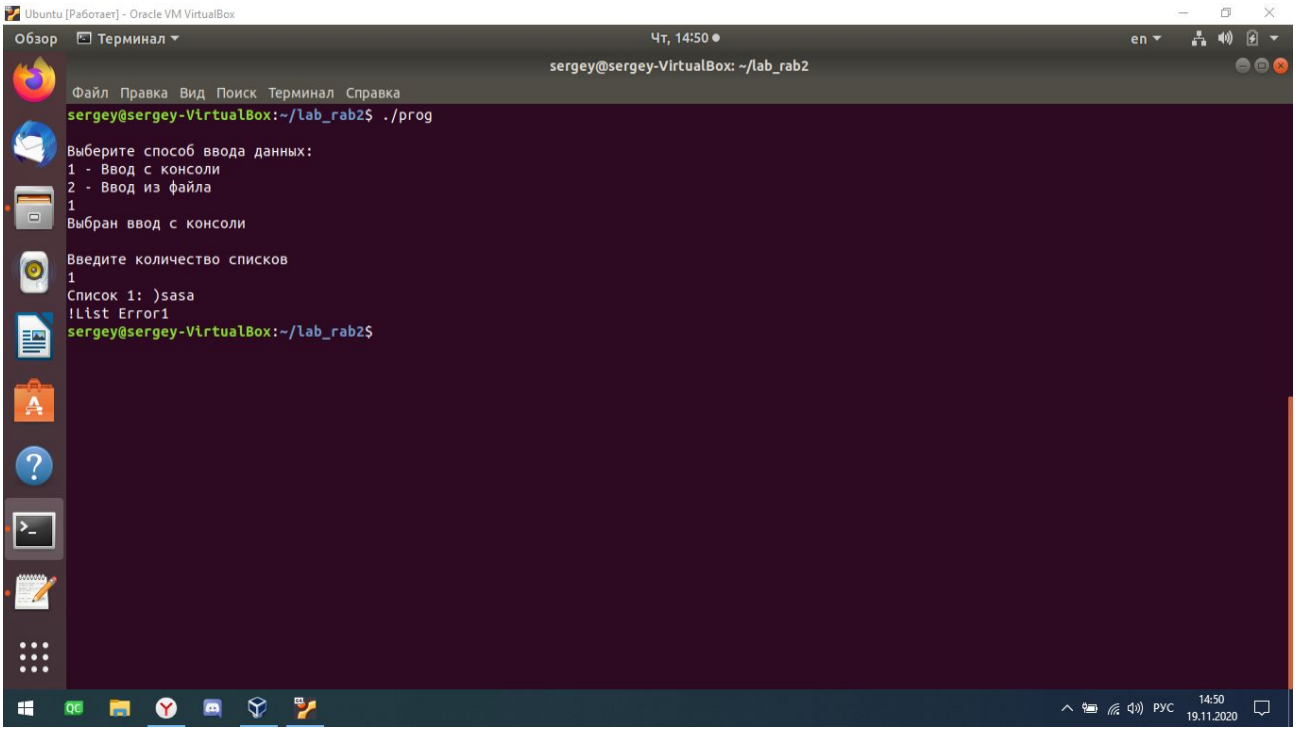
5) Некорректное имя файла



6) Ввод из файла



7) Некорректная запись списка



Вывод.

Была написана программа с использованием иерархических списков и рекурсивных алгоритмов. Были изучены принципы работы с иерархическими списками, взаимодействие с ними при помощи рекурсии, их применение для решения задач.

Приложение А

Исходный код программы

Название файла: list.h

```
#include <iostream>
#include <cstdio>
#include <string.h>
#include <fstream>
using namespace std;

struct s_expr;
typedef struct s_expr LIST;

struct two_ptr{
    LIST* hd;
    LIST* tl;
};

struct s_expr{
    bool tag;
    union {
        char atom;
      two_ptr pair;
    }node;
};

struct list_and_rez{
    LIST* list;
    int rez;
};

typedef struct list_and_rez LIST_AND_REZ;
typedef struct two_ptr TWO_PTR;

LIST* head(const LIST* s);
LIST* tail(const LIST* s);
```

```
LIST* cons(const LIST* h, const LIST* t);
LIST* make_atom(const char x);
bool is_Atom(const LIST* s);
bool is_Null(const LIST* s);
void destroy(LIST* s);
LIST* copy_list(const LIST* x);
void write_list(const LIST* x, ostream* stream);
void write_seq(const LIST* x, ostream* stream);
void read_seq(LIST*& y, istream* stream);
void read_s_expr(char prev, LIST*& y, istream* stream);
void read_list(LIST*& y, int* file_end_flag, istream* stream);
```

Название файла: list.cpp

```
#include "list.h"

LIST* head(const LIST* s){//Функция возвращает указатель на голову списка,
если он не атом
    if(s!=NULL){
      if(!is_Atom(s)){
          return s->node.pair.hd;
      }
      else{
          cerr<<"Error: Head(Atom)\n";
          exit(1);
      }
    }
    else{
        cerr<<"Error: Head(Nil)\n";
      exit(1);
    }
}

LIST* tail(const LIST* s){//Функция возвращает указатель на хвост списка,
если он не атом
    if(s!=NULL){
        if(!is_Atom(s)){
```

```
            return s->node.pair.tl;
        }
        else{
            cerr<<"Error: Tail(Atom)\n";
            exit(1);
        }
    }
    else{
        cerr<<"Error: Tail(Nil)\n";
        exit(1);
    }
}

LIST* cons(LIST* h, LIST* t){//Функция создаёт пару
    LIST* p;
    if(is_Atom(t)){
      cerr<<"Error: cons(*,atom)\n";
      exit(1);
    }
    else{
      p = new LIST;
      if(p==NULL){
          cerr<<"Memory error\n";
          exit(1);
      }
      else{
          p->tag = false;
          p->node.pair.hd = h;
          p->node.pair.tl = t;
          return p;
      }
    }
}

LIST* make_atom(const char x){//Функция создания атома
    LIST* s = new LIST;
```

```
    s->tag = true;
    s->node.atom = x;
    return s;
}

bool is_Atom(const LIST* s){//Функция проверки списка на атомарность
    if(s==NULL){
      return false;
    }
    else{
        return s->tag;
    }
}

bool is_Null(const LIST* s){//Функция проверки списка на пустоту
    return s==NULL;
}

void destroy(LIST* s){//Функция уничтожения элемента списка
    if(s!=NULL){
      if(!is_Atom(s)){
          destroy(head(s));
          destroy(tail(s));
      }
      delete s;
    }
}

LIST* copy_list(const LIST* x){//Функция создания копии списка
    if(is_Null(x)){
      return NULL;
    }
    else if(is_Atom(x)){
      return make_atom(x->node.atom);
    }
    else{
```

```
      return cons(copy_list(head(x)), copy_list(tail(x)));
    }
}

void write_list(const LIST* x, ostream* stream){//Функция печати списка
    if(is_Null(x)){
      *stream<<"()";
    }
    else if(is_Atom(x)){
      *stream<<x->node.atom;
    }
    else{
      *stream<<"(";
      write_seq(x, stream);
      *stream<<")";
    }
}

void write_seq(const LIST* x, ostream* stream){//Вспомогательная функция
печати списка
    if(!is_Null(x)){
        write_list(head(x), stream);
      write_seq(tail(x), stream);
    }
}

void read_list(LIST*& y, int* file_end_flag, istream* stream){//Функция
запуска рекурентной функции считывания списка
    char x;
    do stream->get(x);while(x==' ');
    if(stream->eof()){
      *file_end_flag = 0;
    }
    else if(x=='\n'){
      y = nullptr;
    }
```

```
    else{
      read_s_expr(x,y, stream);
      while(x!='\n'){
          stream->get(x);
      }
    }
}

void read_s_expr(char prev, LIST*& y, istream* stream){//Рекурсивная функция
считывания списка
    if(prev==')'){
      cerr<<"!List Error1"<<endl;
      exit(1);
    }
    else if(prev!='('){
      y = make_atom(prev);
    }
    else{
      read_seq(y, stream);
    }
}

void read_seq(LIST*& y, istream* stream){//Рекурсивная функция считывания
списка
    char x;
    LIST* p1;
    LIST* p2;
    if(stream->eof()){
      cerr<<"!List Error2"<<endl;
      exit(1);
    }
    else{
      stream->get(x);
      while(x==' '){stream->get(x);}
      if(x==')'){
          y = nullptr;
```

```
      }
      else{
          read_s_expr(x, p1, stream);
          read_seq(p2, stream);
          y = cons(p1,p2);
      }
    }
}
```

Название файла: main.cpp

```
#include "list.h"

int input_num(string message){
    int num = 0;
    cout<<message<<'\n';
    char* input = new char[10];
    fgets(input, 10, stdin);
    while(sscanf(input, "%d", &num)!=1){
        cout<<"Ввод некорректный!\n"<<message<<'\n';
      fgets(input, 10, stdin);
    }
    delete [] input;
    return num;
}

void data_save(LIST_AND_REZ* data_mass, int count_of_list){//сохранение
данных в файл
    char* file_name = new char[256];
    cout<<"Введите имя файла сохранения\n";
    cin>>file_name;
    getchar(); //вытаскиваем символ переноса строки из потока
    fstream output_file;
    output_file.open(file_name, fstream::out | fstream::app);//открытие или
создание файла на запись
    for(int i=0;i<count_of_list;i++){
      write_list(data_mass[i].list, &output_file);
      output_file<<" Глубина: "<<data_mass[i].rez<<'\n';
```

```
    }
    delete [] file_name;
    output_file.close();
}

void clear_memory(LIST_AND_REZ* data_mass, int count_of_list){//функция
очистки памяти
    for(int i = 0; i<count_of_list; i++){
        destroy(data_mass[i].list);
    }
    delete [] data_mass;
}

int rec_func_depth(LIST* list, int mid_depth){
    //промежуточные данные
    if(is_Atom(list)){
      cout<<list->node.atom<<" на глубине "<<mid_depth<<'\n';
      return 0;
    }
    if(is_Null(list)){
      return 0;
    }
    int hd_depth = 1+ rec_func_depth(head(list), mid_depth+1);
    int tl_depth = rec_func_depth(tail(list), mid_depth);
    if(hd_depth>tl_depth){
      return hd_depth;
    }
    else{
      return tl_depth;
    }
}

int data_analis(LIST_AND_REZ* data_mass, int count_of_list){

cout<<"-----------------------------------------------------------------
---------------------\nИсходные данные:\n";
```

```
    string dialog_text="\nВыберите дальнейшее действие:\n1 - сохранить
данные в файл и продолжить\n2 - сохранить данные в файл и выйти\n3 -
продолжить без сохранения\n4 - выйти без сохранения";
    for(int i = 0; i<count_of_list;i++){
        write_list(data_mass[i].list, &cout);
        cout<<'\n';
    }
    cout<<"\nПромежуточные данные алгоритма:\n\n";
    for(int i = 0; i<count_of_list;i++){
      cout<<"Рассчёт глубины для списка : ";
      write_list(data_mass[i].list, &cout);
      cout<<'\n';
      data_mass[i].rez = rec_func_depth(data_mass[i].list, 0);
      cout<<"Итоговая глубина: "<<data_mass[i].rez<<"\n\n";
    }
    cout<<"Нажмите ENTER, чтобы продолжить";
    getchar();
    while(1){
        switch (input_num(dialog_text)){ //выбор дальнейших действий
пользователем
          case 1: data_save(data_mass,
count_of_list);clear_memory(data_mass, count_of_list);return 1;
break;//сохранение и очистка данных
          case 2: data_save(data_mass,
count_of_list);clear_memory(data_mass, count_of_list);return 0; break;
          case 3: clear_memory(data_mass, count_of_list);return 1; break;
          case 4: clear_memory(data_mass, count_of_list);return 0; break;
          default: cout<<"Команда не распознана!\n";break;
        }
    }
}

int console_input(){
    string message = "Введите количество списков";
    int count_of_list = input_num(message);
    if (count_of_list<=0){
```

```
        return 1;
    }
    LIST_AND_REZ* data_mass = new LIST_AND_REZ[count_of_list];
    for(int i = 0; i<count_of_list; i++){
      cout<<"Список "<<i+1<<": ";
      read_list(data_mass[i].list, nullptr, &cin);//считывание строки из
консоли
    }
    if(data_analis(data_mass, count_of_list)){//вызов функции анализа данных
        return 1;
    }
    return 0;
}

int file_input(){
    int count_of_list = 0;
    int correct_file_name_flag = 0;  //флаг корректного имени файла ввода
    fstream file_input;
    char* file_name = new char[256];
    while(!correct_file_name_flag){//цикл до ввода корректного имени файла
        cout<<"Введите имя файла\n\n";
        cin>>file_name;
        file_input.open(file_name, fstream::in);//открывается файл ввода
      if(file_input.is_open()){
          correct_file_name_flag = 1;
      }
      else{
          cout<<"\nФайла с таким именем не найдено!\n";
          memset(file_name, '\0', 256);
      }
    }
    getchar();//убираем символ переноса строки из потока ввода
    delete [] file_name;
    int buff = 10;//буффер количества строк
    int file_end_flag = 1;//флаг конца файла
    LIST_AND_REZ* data_mass = new LIST_AND_REZ[buff];
```

```
    LIST_AND_REZ* rezerv_data_mass;
    while(file_end_flag){
      if(count_of_list==buff){//проверка на заполнение буффера
          buff+=10;
          rezerv_data_mass = new LIST_AND_REZ[buff];
          for(int i = 0;i<buff-10;i++){
            rezerv_data_mass[i].list = data_mass[i].list;
          }
          delete [] data_mass;
             data_mass = rezerv_data_mass;
             rezerv_data_mass = nullptr;
      }
      //считывание строки из файла
      read_list(data_mass[count_of_list].list, &file_end_flag, &file_input);
      count_of_list++;
    }
    file_input.close();
    delete data_mass[count_of_list-1].list;
    if(data_analis(data_mass, count_of_list-1)){//вызов функции анализа
данных
        return 1;
    }
    return 0;
}

int main(){
    string start_dialog = "\nВыберите способ ввода данных:\n1 - Ввод с
консоли\n2 - Ввод из файла\n3 - Выйти из программы";
    while(1){
      switch(input_num(start_dialog)){
      case 1:
          cout<<"Выбран ввод с консоли\n\n";
          if(!console_input()){
              return 0;
          }
          break;
```

```
        case 2:
            cout<<"Выбран ввод из файла\n\n";
            if(!file_input()){
                return 0;
            }
            break;
        case 3:
            cout<<"Выход из программы\n";
            return 0;
            break;
        default:
            cout<<"Ответ некорректный!\n\n";
        }
    }
}
```