**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра информационных систем**

отчет

**по практической работе №2**

**по дисциплине «Программирование»**

Тема: **Линейные структуры данных. Динамические массивы и двусвязные списки**

|  |  |  |
| --- | --- | --- |
| Студент гр. 9894 |  | Алексеев В.А. |
| Преподаватель |  | Глущенко А.Г. |

Санкт-Петербург

2020

**Цель работы.**

Изучение свойств и организация динамических массивов и двусвязных списков. Получение практических навыков в работе с динамическими массивами и двусвязными списками. Проведение сравнительной характеристики скорости вставки, получения и удаления элементов из них.

**Основные теоретические положения.**

1. **Распределение памяти при работе программы**

Схема распределения памяти под программу показана на рис. 1.

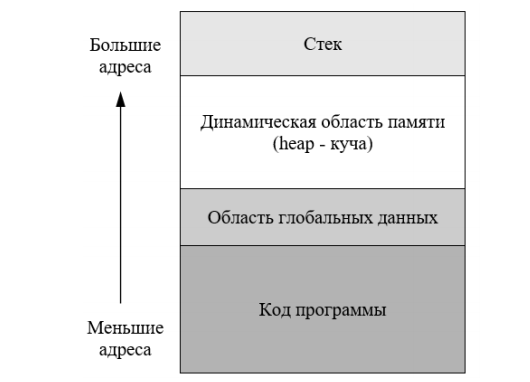


Рисунок 1 – Схема распределения памяти под программу

Область кода программы предназначена для хранения инструкций функций программы, обеспечивающих обработку данных. Данные в программе представляются переменными и константами. Для хранения глобальных данных предназначена область глобальных данных. Стек программы используется при вызове функций для передачи параметров и хранения локальных данных.

Распределение памяти для хранения всех обычных переменных осуществляется компилятором, и адреса, и объемы соответствующих участков памяти (в области 85 глобальных данных) жестко закреплены за этими переменными на все время работы программы и изменено быть не может.

Однако во многих задачах невозможно заранее предсказать, сколько места (количество переменных, объемы массивов и т.д.) потребуется для решения задачи – это так называемые задачи с неопределенной размерностью. Решить эту проблему можно лишь в том случае, если иметь механизм, позволяющий создавать новые объекты по мере возникновения необходимости в этих объектах или изменять объемы памяти, выделенные под эти объекты (например, объемы массивов).

Между областью глобальных данных и стеком располагается так называемая динамическая область памяти, которую как раз и можно использовать в процессе работы программы для реализации механизма динамического управления памятью.

1. **Одномерные динамические массивы**

Для того чтобы создать в динамической области некоторый объект необходима одна обычная (не динамическая переменная) переменная-указатель. Сколько таких объектов нам понадобится для одновременной обработки – столько необходимо иметь обычных переменных-указателей. Таким образом, проблема “задач неопределенной размерности” созданием одиночных динамических объектов решена быть не может.

Решить эту проблему поможет возможность создавать в динамической области памяти массивы объектов с таким количеством элементов, которое необходимо в данный момент работы программы – то есть создание динамических массивов. Действительно, для представления массива требуется всего одна переменнаяуказатель, а в самом массиве, на который ссылается этот указатель, может быть столько элементов, сколько требуется в данный момент времени.

Для создания одномерного динамического массива, элементами которого являются, например, действительные числа, используется следующий синтаксис инструкции new (стиль С++):

double \*Arr = new double [100];

Освободить динамическую область от этого массива можно с помощью инструкции delete.

delete [] Arr;

После чего занятый участок памяти будет возвращен в список свободной памяти и может быть повторно использован для размещения других динамических объектов.

Язык C++ поддерживает и “старый”, заимствованный от языка C, стиль работы с динамической областью. Достаточно часто бывает полезно использовать именно этот механизм управления динамической памятью.

В языке C отсутствуют инструкции new и delete. Вместо них для управления динамической памятью используются библиотечные функции:

void \*malloc (size );

void \*calloc(num, size );

void free( void \*memblock );

void \*realloc( void \*memblock, size );

Функция malloc выделяет в динамической области size байт памяти и возвращает адрес этого участка в виде указателя (void \*). Поскольку возвращаемый указатель не привязан ни к какому типу данных, при работе с ним потребуется явное приведение типов данных (см. пример ниже).

Функция calloc выделяет в динамической области size \* num байт памяти и возвращает адрес этого участка в виде указателя (void \*).

Функция free освобождает участок динамической памяти по адресу memblock и возвращает его в список свободной памяти для повторного использования.

Функция realloc позволяет изменить размер (уменьшить или увеличить) ранее выделенной по адресу memblock памяти, установив новый размер выделенного участка равным size байт. При увеличении размера выделенного участка данные, которые хранились в старом участке, копируются в новый участок памяти. При уменьшении объема выделенного участка, данные которые хранились в нем, усекаются до нового размера. Функция возвращает указатель на область памяти нового размера.

Работа с одномерным динамическим массивом осуществляется так же, как и с обычным. При этом стиль использования динамических массивов С имеет весомое преимущество над С++, которое заключается в изменении размерности массива. Дело в том, что в C++ нет функций увеличения размерности. Увеличить размер массива можно, создав новый динамический массив нужной размерности, скопировав данные из старого массива в новый и освободив память от старого массива.

1. **Двумерные динамические массивы**

Создание многомерных массивов несколько сложнее, нежели одномерных динамических массивов. Условно двумерный динамических массив можно представить, как на рис. 2.

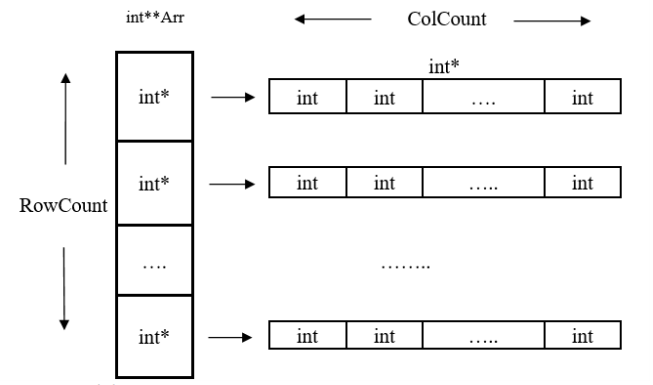


Рис 2 – Представление двумерного динамического массива

Чтобы получить двумерный массив, необходимо:

1) создать одномерный динамический массив из RowCount указателей на базовый тип элементов массива (в нашем случае указателей на тип int);

2) в цикле создать RowCount одномерных динамических массивов, каждый из которых содержит ColCount элементов базового типа (в нашем случае указателей на тип int) и адреса их первых элементов записать в соответствующие элементы “вертикального” массива.

Обычный одномерный массив определяется как указатель на базовый тип данных элементов этого массива. Базовым типом элементов этого массива являются указатели int\*. Для того чтобы определить указатель на указатель достаточно использовать следующую конструкцию: (int\*)\* или проще int\*\*.

Таким образом, для того чтобы создать динамический массив Arr из указателей, можно поступить так:

int \*\* Arr = new int \* [ RowCount ];

Тогда для создания всего двумерного динамического массива необходимо выполнить следующие действия:

int \*\* Arr = new int \* [ RowCount ]; // Создаем “вертикальный” массив

for ( int i = 0; i < RowCount; ++ i )

Arr [ i ] = new int [ ColCount ]; // Создаем i-ый массив-строку

Для освобождения памяти необходимо:

1. сначала в цикле удалить RowCount массивов-строк;

2. затем удалить “вертикальный” массив.

for ( int i = 0; i < RowCount; ++ i )

delete [ ] Arr [ i ]; // Удаляем i-ый массив-строку

delete [ ] Arr; // Удаляем “вертикальный” массив

Для создания динамических двумерных массивов с другими базовыми типами элементов достаточно в предыдущих примерах заменить тип данных int, на необходимый тип данных. Ну, и конечно, изменить работу с элементами массива в соответствии с их типом данных. Обязательные места исправлений выделены красным цветом.

По аналогии с двумерными динамическими массивами можно создавать и массивы большей мерности.

**4. Односвязные линейные списки**

Одномерный однонаправленный список представляет собой совокупность отдельных элементов, каждый из которых содержит две части – информационную (𝐷𝑎𝑡𝑎) и адресную (𝑇𝑎𝑖𝑙). Информационная часть предназначена для хранения “полезных” данных и может иметь практически любой тип. Адресная часть каждого элемента содержит адрес следующего элемента списка. Схематическое изображение такого списка представлено на рис. 3.

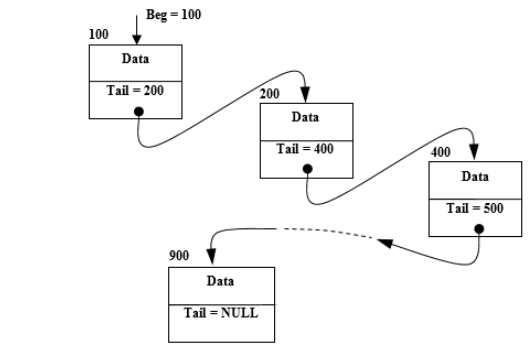


Рисунок 3 – Схематическое изображение односвязного списка

Для работы со списком достаточно знать только адрес первого элемента списка (𝐵𝑒𝑔). Зная адрес первого элемента списка можно последовательно получить доступ к любому другому его элементу.

Поскольку каждый элемент списка должен иметь две части, логичнее всего представить его в виде следующей структуры:

struct list

{

int data;

list \*tail;

};

Типовыми операциями при работе со списками являются:

• создание списка;

• освобождение памяти от списка (удаление списка);

• доступ к заданному элементу списка для манипуляций с его информационной частью;

• добавление нового элемента к списку;

• удаление элемента из списка;

• перестановка элемента списка на новую позицию внутри списка.

Достоинством подобных структур являются простота добавления, удаления и перестановки элементов списка, которые осуществляются путем манипуляций с адресными частями без перезаписи всего списка.

**5. Удаление, добавление и обмен элементов односвязного списка**

На рис. 4 представлена схема удаления первого элемента из односвязного списка.

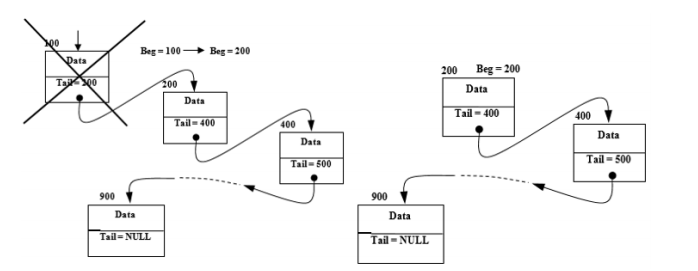


Рис. 4 – Схема удаления первого элемента из односвязного списка

Если стоит задача удалить первый элемент связного списка, то необходимо изменить адрес первого элемента списка на адрес следующего за первым элемента и освободить динамическую память от первого элемента.

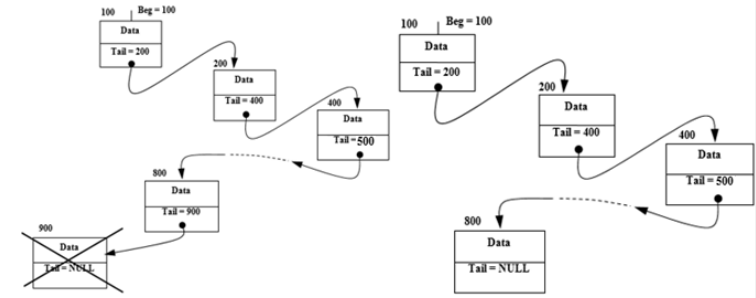
Схожий принцип действия и у удаления последнего элемента (рис. 5).

Рисунок 5 – Удаление последнего элемента списка

Сначала необходимо изменить адресную часть 𝑁 − 1 элемента так, чтобы теперь он имел нулевой указатель, затем нужно освободить память 𝑁-го от элемента.

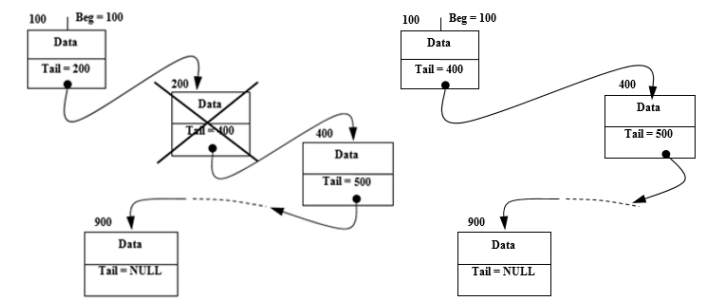
С удалением элемента из середины списка (рис. 6) дело обстоит практически так же, как и с удалением с конца списка.

Рисунок 6 –Удаление элемента из середины списка

Вставка и обмен элементов односвязного списка происходит по схожему принципу. Более подробно эти операции будут рассмотрены на двусвязных списках.

**6. Двусвязные линейные списки**

Одним из недостатков односвязных списков является то, что узел (элемент списка) имеет указатель только на следующий элемент. Вернутся из текущего элемента к предыдущему явным способом невозможно.

Каждый узел двусвязного (двунаправленного) линейного списка содержит два поля указателей – на следующий и на предыдущий узлы. Указатель на предыдущий узел корня списка содержит нулевое значение. Указатель последнего узла также содержит нулевое значение.

Поскольку каждый элемент списка должен иметь три части, логичнее всего представить его в виде следующей структуры:

struct list

{

int data;

list \*head;

list \*tail;

};

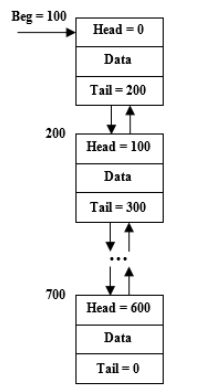
На рис. 7 показано схематическое представление двусвязного списка. Поле 𝐻𝑒𝑎𝑑 содержит адрес предыдущего элемента, поле 𝑇𝑎𝑖𝑙 содержит адрес следующего элемента списка. Такая организация списка позволяет перемещаться по его элементам в двух направлениях.

Рисунок 7 – Схематическое изображение двусвязного списка

Основные действия, производимые над узлами ДЛС (двусвязного линейного списка):

• инициализация списка;

• добавление узла в список;

• удаление узла из списка;

• удаление корня списка;

• вывод элементов списка;

• вывод элементов списка в обратном порядке;

• взаимообмен двух узлов списка.

Порядок действия очень похож на односвязный линейный список, но необходимо учитывать то, что в двусвязном списке имеется два указателя: на следующий и предыдущий элементы.

**7. Удаление, вставка, обмен элементов двусвязного списка**

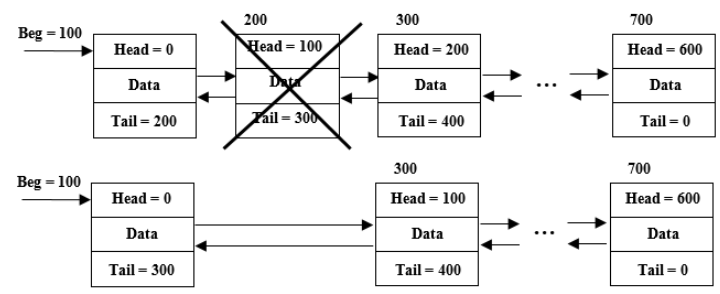
Удаление первого элемента и последнего элемента практически аналогично удалению элемента из односвязного списка. Нужно изменить значение указателя на первый элемент, обнулить значение указателя головы/хвоста, освободить динамическую память. С удалением элемента из середины списка (рис. 8) дело обстоит сложнее – необходимо проделать аналогичные операции по отношению к двум узлам, а не к одному.

Рисунок 8 – Удаление узла из середины двусвязного списка

Чтобы удалить узел из середины списка необходимо изменить указатели 𝐻𝑒𝑎𝑑 и 𝑇𝑎𝑖𝑙 предыдущего и следующего элемента, связав их вместе. Затем нужно освободить динамическую память от удаленного элемента.

Со вставкой нового узла в двусвязный линейный список дело обстоит аналогично. Так же присутствует три ситуации: вставка нового корня, вставка последним элементом и вставка узла в середину списка.

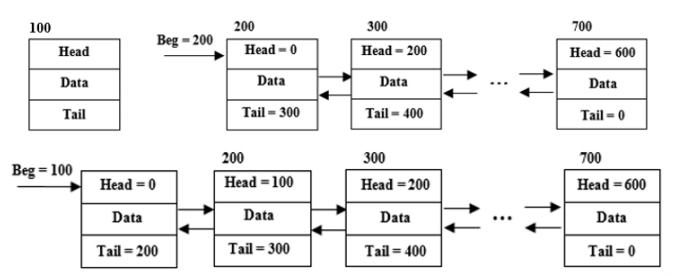
На рис. 9 изображена вставка нового корневого узла. Видно, что от удаления процедура не особо отличается: необходимо изменить 𝐻𝑒𝑎𝑑 указатель прошлого корневого узла, указатель на корневой узел и правильно связать новый узел со списком.

Рисунок 9 – Вставка корневого узла

Вставка узла в конец списка не должна вызывать затруднений: необходимо изменить 𝑇𝑎𝑖𝑙 указатель прежнего последнего элемента на адрес нового узла, затем правильно связать новый узел с двусвязным списком (𝐻𝑒𝑎𝑑 – адрес на предыдущий элемент, 𝑇𝑎𝑖𝑙 – NULL указатель). Вставка же узла в середину списка представлена на рис. 10.

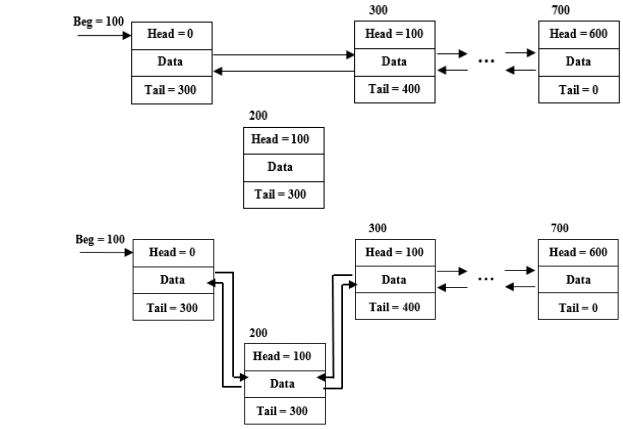


Рисунок 10 – Вставка узла в середину двусвязного списка

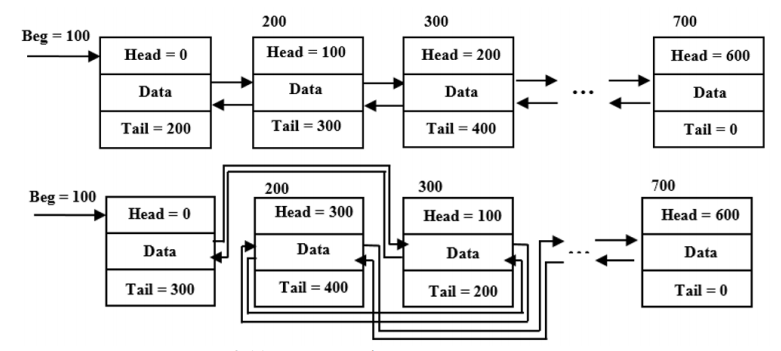
Больше всего трудностей может возникнуть именно с взаимообменом элементов (рис 11) двусвязного списка. Главная проблема – возможная путаница в указателях. Нужно правильно изменить указатели всех затрагиваемых узлов.

Рисунок 11 – Взаимообмен узлов двусвязного списка

Видно, сколько необходимо изменить адресов указателей, чтобы двумерный список работал корректно. Если хоть один указатель будет ссылаться неправильно, то список будет работать некорректно и может вовсе зациклиться. Причем программист этого может не заметить, если поле 𝐷𝑎𝑡𝑎 однородно или вовсе отсутствует. Поэтому (как и с любыми указателями) нужно быть внимательным.

**Постановка задачи.**

Необходимо реализовать программу, которая выполняет следующие действия:

1) Формирование целочисленного одномерного массива размерности N, где:

a) пользователь вводит количество элементов в массиве, который будет автоматически заполняться случайными числами (0 до 99);

б) пользователь вводит в консоль элементы массива, N определяется автоматически по количеству введенных элементов;

в) массив считывается с файла, N определяется как количество элементов массива в файле.

2) Определение скорости создания динамического массива п. 1.

3) Вставка, удаление и получение элемента массива. Удаление и получение элемента необходимо реализовать по индексу и по значению.

4) Определение скорости вставки, удаления и получения элемента массива п. 3.

5) Формирование двусвязного списка размерности N, где:

a) пользователь вводит количество элементов в списке, который будет автоматически заполняться случайными числами (0 до 99);

б) пользователь вводит в консоль элементы списка, N определяется автоматически по количеству введенных элементов;

в) список считывается с файла, N определяется как количество элементов списка в файле.

6) Определение скорости создания двусвязного списка п. 6.

7) Вставка, удаление и получение элемента двусвязного списка. Удаление и получение элемента необходимо реализовать по индексу и по значению.

8) Определение скорости вставки, удаление и получения элемента двусвязного списка п. 8.

9) Реализовать и сравнить скорость работы bubble sort для исходного динамического массива и двусвязного списка.

**Выполнение работы.**

1. Реализация условной оболочки (приложение Д) для упрощения взаимодействия с массивом. Эта оболочка подчиняется (то есть наследует все методы) интерфейсу IStructure из приложения В, а значит реализует все подпункты в п. 1.

Таким образом управляющий объект – интерпретатор команд (приложение Б), может взаимодействовать с массивом, опираясь на интерфейс IStructure.

1. Реализация двусвязного списка и его оболочки (приложение Г). Эта оболочка также подчиняется интерфейсу ISctructure, а значит управляющий объект CommandInterpreter может взаимодействовать со списком так же, как и с массивом.

**Выводы.**

Приложение А

Заголовочный Файл

#include "CommandInterpreter.h"

int main() {

CommandInterpreter\* shell = new CommandInterpreter();

shell->Start();

delete(shell);

return 0;

}

Приложение Б

Интерпретатор команд

#pragma once

#include <iostream>

#include <string>

#include <vector>

#include "List.h"

#include "Array.h"

#include "IStructure.h"

#include <ctime>

#include <functional>

typedef void (IStructure::\* MethodPtr)(int); // тайпдей на указатель на константный метод класса IStructure

class CommandInterpreter

{

private:

Array \_vector;

List \_linkedList;

// Commands's values as string

const char\* \_array = "array";

const char\* \_list = "list";

const char\* \_fillRand = "fill\_rand";

const char\* \_fillHand = "fill\_hand";

const char\* \_fillFile = "fill\_file";

const char\* \_insert = "insert";

const char\* \_remove = "remove";

const char\* \_removevalue = "removevalue";

const char\* \_get = "find";

const char\* \_getvalue = "findvalue";

const char\* \_print = "print";

const char\* \_sort = "sort";

// Commands's values as string as enum's elements

static enum Commands

{

Array,

List,

FillRand,

FillByHand,

FillByFile,

Print,

Insert,

Remove,

RemoveValue,

GetValue,

Get,

Sort,

Unrecognized

};

void ErrorMessage(const char\* message)

{

std::cout << ">>> ERROR " << message << std::endl;

}

// Define enum element by string value

int InterpreteCommand(std::string command)

{

if (command == \_array) return Array;

if (command == \_list) return List;

if (command == \_fillRand) return FillRand;

if (command == \_fillHand) return FillByHand;

if (command == \_fillFile) return FillByFile;

if (command == \_insert) return Insert;

if (command == \_remove) return Remove;

if (command == \_get) return Get;

if (command == \_print) return Print;

if (command == \_getvalue) return GetValue;

if (command == \_removevalue) return RemoveValue;

if (command == \_sort) return Sort;

return Unrecognized;

}

// Strategy definer

// command contains commandName[0] and arguments [1], [2], [3], ...

bool DefineStrategy(std::vector<std::string> command)

{

IStructure\* structure = nullptr;

int structChoice = InterpreteCommand(command[0]);

if (command.size() < 1 || structChoice == Unrecognized)

{

ErrorMessage("missed argument");

return false;

}

if (structChoice == Array) structure = &\_vector;

else structure = &\_linkedList;

int strategie = InterpreteCommand(command[1]);

switch (strategie)

{

case FillRand:

{

if (command.size() < 3) ErrorMessage("missed argument");

else

{

unsigned int start = clock();

int size = std::atoi(command[2].c\_str());

structure->create\_and\_fill(size);

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case FillByHand:

{

if(command.size() < 4) ErrorMessage("missed argument");

else

{

int\* values = new int[command.size() - 3];

for (int i = 3; i < command.size(); ++i) values[i - 3] = std::atoi(command[i].c\_str());

unsigned int start = clock();

structure->set(std::atoi(command[2].c\_str()), values);

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case FillByFile:

{

if(command.size() < 3) ErrorMessage("missed argument");

else

{

unsigned int start = clock();

structure->create\_from\_file(command[2]);

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case Insert:

{

if(command.size() < 4) ErrorMessage("missed argument");

else

{

unsigned int start = clock();

structure->insert(std::atoi(command[2].c\_str()), std::atoi(command[3].c\_str()));

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case Remove:

{

if (command.size() < 3)ErrorMessage("missed argument");

else

{

unsigned int start = clock();

structure->remove(std::atoi(command[2].c\_str()));

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case RemoveValue:

{

if (command.size() < 3)ErrorMessage("missed argument");

else

{

unsigned int start = clock();

structure->removevalue(std::atoi(command[2].c\_str()));

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case Get:

{

if (command.size() < 3)ErrorMessage("missed argument");

else

{

unsigned int start = clock();

std::cout << ">>> " << structure->find(std::atoi(command[2].c\_str())) << std::endl;

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case GetValue:

{

if (command.size() < 3)ErrorMessage("missed argument");

else

{

unsigned int start = clock();

structure->findvalue(std::atoi(command[2].c\_str()));

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

}

break;

}

case Sort:

{

unsigned int start = clock();

structure->sort();

unsigned int end = clock();

std::cout << ">>> delay - " << end - start << std::endl;

break;

}

case Print:

{

structure->print();

break;

}

case Unrecognized:

{

ErrorMessage("unrecognized command");

}

}

return false;

}

public:

// Start command interpretter

void Start()

{

std::vector<std::string> fullCommand;

while (true)

{

std::cout << ">>> ";

std::string currCommand;

std::getline(std::cin, currCommand);

if (currCommand.size() == 0) std::getline(std::cin, currCommand);

std::string commandPart;

for (int i = 0; i < currCommand.length(); ++i)

{

if (currCommand[i] == ' ' && !commandPart.empty())

{

fullCommand.push\_back(commandPart);

commandPart.clear();

}

else

{

commandPart.push\_back(currCommand[i]);

}

}

fullCommand.push\_back(commandPart);

if (DefineStrategy(fullCommand)) break;

fullCommand.clear();

}

}

~CommandInterpreter() {}};

Приложение В

интерфейс структур

#pragma once

#include <string>

// Interface for list and array

class IStructure

{

public:

virtual void create\_and\_fill(int size) = 0;

virtual void create\_from\_file(std::string path) = 0;

virtual void set(int size, int\* values) = 0;

virtual void insert(int id, int value) = 0;

virtual void remove(int id) = 0;

virtual void removevalue(int value) = 0;

virtual int find(int id) = 0;

virtual void findvalue(int value) = 0;

virtual void print() = 0;

virtual void sort() = 0;

};

Приложение г

двусвязный список

#pragma once

#include <iostream>

#include <ctime>

#include <fstream>

#include <string>

#include "IStructure.h"

// Node structure as a main element of list

struct Node

{

int value;

Node\* prev = nullptr;

Node\* next = nullptr;

};

class List : public IStructure

{

private:

Node\* \_begin = nullptr;

Node\* \_end = nullptr;

int \_size;

// Drop method: deletes all nodes in list

void drop()

{

Node\* curr = \_begin;

while (curr != nullptr)

{

Node\* tmp = curr->next;

delete(curr);

curr = tmp;

}

\_begin = \_end = nullptr;

}

public:

List()

{

\_size = 0;

srand(time(NULL));

}

// Add a new node with value from arguments to the back of the list

void push\_back(int value)

{

Node\* node = new Node();

node->value = value;

if (\_begin == nullptr && \_end == nullptr)

{

\_end = \_begin = node;

}

else

{

\_end->next = node;

node->prev = \_end;

\_end = node;

}

\_size++;

}

// Add a new node with value from arguments to the front of the list

void push\_front(int value)

{

Node\* node = new Node();

node->value = value;

if (\_begin == nullptr && \_end == nullptr)

{

\_end = \_begin = node;

}

else

{

\_begin->prev = node;

node->next = \_begin;

node->prev = nullptr;

\_begin = node;

}

\_size++;

}

// Delete one element from the back of the list

void pop\_back()

{

if (\_begin == nullptr && \_end == nullptr)

return;

if (\_begin == \_end && \_begin != nullptr)

{

delete(\_begin);

\_begin = \_end = nullptr;

}

else

{

\_end->prev->next = nullptr;

Node\* toDelete = \_end;

\_end = \_end->prev;

delete(toDelete);

}

\_size--;

}

// Delete one element from the front of the list

void pop\_front()

{

if (\_begin == nullptr && \_end == nullptr)

return;

if (\_begin == \_end && \_begin != nullptr)

{

delete(\_begin);

\_begin = \_end = nullptr;

}

else

{

\_begin->next->prev = nullptr;

Node\* toDelete = \_begin;

\_begin = \_begin->next;

delete(toDelete);

}

\_size--;

}

// Inirialize a list with random values

void create\_and\_fill(int size)

{

if(\_begin != nullptr) drop();

for (int i = 0; i < size; ++i)

push\_back(rand() % 100);

}

// Setter for list

void set(int size, int\* values)

{

if (\_begin != nullptr) drop();

\_size = size;

for (int i = 0; i < \_size; ++i)

push\_back(values[i]);

}

// Initialize a list with file, which contain values

void create\_from\_file(std::string path)

{

if (\_begin != nullptr) drop();

std::ifstream\* stream = new std::ifstream(path);

std::string buff;

int counter = 0;

while (!stream->eof())

{

std::getline(\*stream, buff);

push\_back(std::atoi(buff.c\_str()));

counter++;

}

\_size = counter;

delete(stream);

}

// Return value of element with corresponding id

int find(int id)

{

if (id > \_size || id < 0) return INT\_MIN;

Node\* curr = \_begin;

for(int i = 1; i < id; ++i)

{

curr = curr->next;

}

return curr->value;

}

// Find and print all elements with corresponding value

void findvalue(int value)

{

Node\* curr = \_begin;

int counter = 0;

while (curr != nullptr)

{

counter++;

if (curr->value == value)

{

std::cout << " id - " << counter << std::endl;

}

curr = curr->next;

}

}

// Insert a new element with corresponding id

void insert(int id, int value)

{

if (id > \_size) return;

if (id == 0)

{

push\_front(value);

return;

}

if (id == \_size)

{

push\_back(value);

return;

}

Node\* toInsert = new Node();

toInsert->value = value;

Node\* curr = \_begin;

for (int i = 1; i < id - 1; ++i)

{

curr = curr->next;

}

curr->next->prev = toInsert;

toInsert->next = curr->next;

toInsert->prev = curr;

curr->next = toInsert;

\_size++;

}

// Remove element with corresponding id

void remove(int id)

{

if (id > \_size || id < 0) return;

if (id == 1)

{

pop\_front();

return;

}

if (id == \_size)

{

pop\_back();

return;

}

Node\* curr = \_begin;

for (int i = 1; i < id - 1; ++i)

{

curr = curr->next;

}

Node\* tmp = curr->next;

curr->next->next->prev = curr;

curr->next = curr->next->next;

delete(tmp);

\_size--;

}

// Remove ALL values with corresponding value

void removevalue(int value)

{

Node\* curr = \_begin;

while (curr != nullptr)

{

if (curr->value == value)

{

if (curr == \_begin)

{

curr = curr->next;

pop\_front();

}

else if (curr == \_end)

{

curr = curr->next;

pop\_back();

}

else

{

curr->prev->next = curr->next;

curr->next->prev = curr->prev;

Node\* tmp = curr;

curr = curr->next;

delete(tmp);

}

}

else

{

curr = curr->next;

}

}

}

// Sort list in ascending order

void sort()

{

Node\* curr = \_begin;

for (int i = 0; i < \_size - 1; i++)

{

Node\* currSecondary = \_begin;;

for (int j = 0; j < \_size - i - 1; j++)

{

if (currSecondary->value > currSecondary->next->value)

{

Swap(&currSecondary->value, &currSecondary->next->value);

}

currSecondary = currSecondary->next;

}

curr = curr->next;

}

}

// Swap two values by address

void Swap(int\* a, int\* b)

{

int buff = \*a;

\*a = \*b;

\*b = buff;

}

// Print whole list

void print()

{

Node\* curr = \_begin;

int counter = 1;

std::cout << "\n";

while (curr != nullptr)

{

Node\* tmp = curr->next;

std::cout << counter << ": " << curr->value << std::endl;

curr = tmp;

counter++;

}

std::cout << "\n";

}

// Destructor for cleaning memory

~List()

{

drop();

}

};

Приложение д

оболочка массива (подобие вектора)

#include <iostream>

#include <ctime>

#include <fstream>

#include <string>

#include "IStructure.h"

class Array : public IStructure

{

private:

int \_init\_value = 10;

int\* \_curr\_array;

int \_curr\_size;

int \_sign\_size;

// Drop array and resize it with corresponding size value

void dropresize(int size)

{

delete[] \_curr\_array;

\_curr\_array = new int[size];

\_curr\_size = size;

\_sign\_size = 0;

}

// Copies array and resize it

void copyresize()

{

int\* new\_arr = new int[\_curr\_size + 10];

for (int i = 0; i < \_sign\_size; ++i)

{

new\_arr[i] = \_curr\_array[i];

}

delete[] \_curr\_array;

\_curr\_array = new\_arr;

\_curr\_size += 10;

}

public:

Array()

{

\_curr\_array = new int[\_init\_value];

\_curr\_size = \_init\_value;

\_sign\_size = 0;

srand(time(NULL));

}

// Create new array instanse if needed and fill it with random values

void create\_and\_fill(int size)

{

dropresize(size);

foreach([](int\* p) -> void { \*p = rand() % 100; });

\_sign\_size = size;

}

// Create a new array and fill it with values from file

void create\_from\_file(std::string path)

{

dropresize(10);

\_curr\_size = 10;

\_sign\_size = 0;

std::ifstream\* stream = new std::ifstream(path);

std::string buff;

int counter = 0;

while (!stream->eof())

{

if (\_sign\_size == \_curr\_size) copyresize();

std::getline(\*stream, buff);

\_curr\_array[counter] = std::atoi(buff.c\_str());

counter++;

\_sign\_size++;

}

delete(stream);

}

// Setter

void set(int size, int\* values)

{

\_curr\_size = size;

\_sign\_size = size;

\_curr\_array = values;

}

// Insert corresponding value with corresponding id

void insert(int id, int value)

{

if (id > \_sign\_size) return;

if (\_sign\_size == \_curr\_size) copyresize();

for (int i = \_sign\_size; i >= id; --i)

{

\_curr\_array[i + 1] = \_curr\_array[i];

}

\_curr\_array[id] = value;

\_sign\_size++;

}

// Removes element with corresponding id

void remove(int id)

{

for (int i = id; i < \_sign\_size - 1; ++i)

{

\_curr\_array[i] = \_curr\_array[i + 1];

}

\_sign\_size--;

}

// Removes ALL values with corresponding value

void removevalue(int value)

{

for (int i = 0; i < \_sign\_size; ++i)

{

if (\_curr\_array[i] == value)

remove(i);

}

}

// Find and return element with corresponding id

int find(int id)

{

if (id <= \_sign\_size - 1 || id < 0) return \_curr\_array[id];

else return INT\_MIN;

}

// Fing and print all element with corresponding values

void findvalue(int value)

{

for (int i = 0; i < \_curr\_size; ++i)

{

if (\_curr\_array[i] == value)

{

std::cout <<" id - " << i << std::endl;

}

}

}

// Iterates though array (just for fun)

void foreach(void(\*func)(int\*))

{

for (int\* i = \_curr\_array; i < \_curr\_array + \_curr\_size; ++i)

func(i);

}

// Print whole array

void print()

{

int counter = 0;

std::cout << "\n";

for (int i = 0; i < \_sign\_size; ++i)

{

std::cout << counter << ": " << \_curr\_array[i] << std::endl;

counter++;

}

std::cout << "\n";

}

// Sort array in ascending order

void sort()

{

for (int i = 0; i < \_sign\_size - 1; i++)

{

for (int j = 0; j < \_sign\_size - i - 1; j++)

{

if (\_curr\_array[j] > \_curr\_array[j + 1])

{

Swap(&\_curr\_array[j], &\_curr\_array[j + 1]);

}

}

}

}

// Swap two values by address

void Swap(int\* a, int\* b)

{

int buff = \*a;

\*a = \*b;

\*b = buff;

}

// Destructor for cleaning memory

~Array()

{

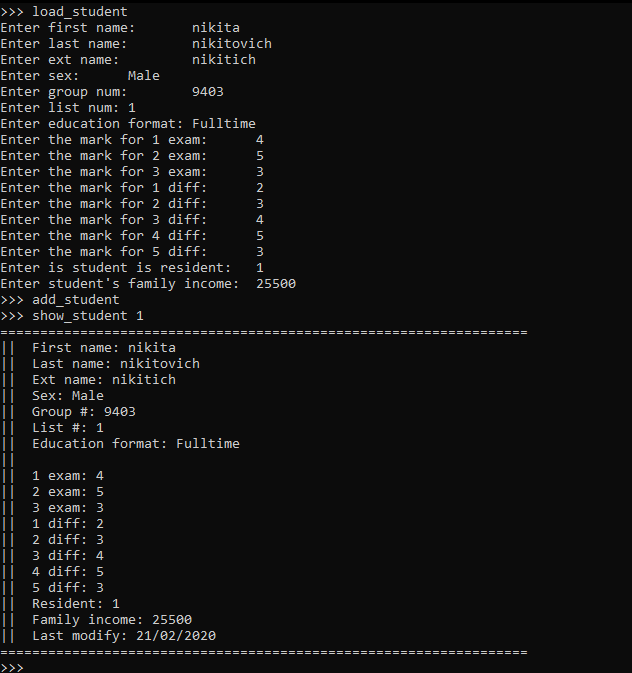
delete[] \_curr\_array;

}

};

Приложение ж

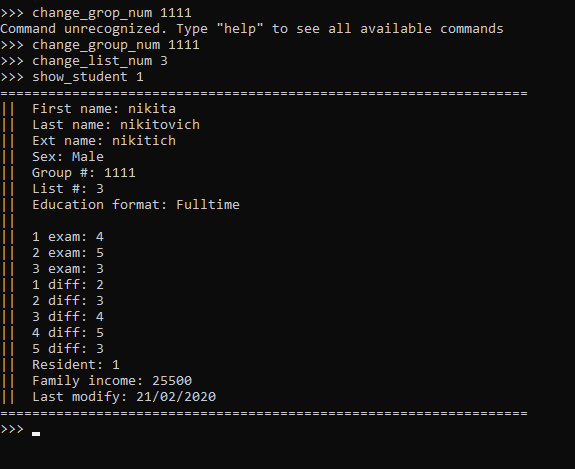
изображение 1



Изображение 1 – демонстрация работы первого пункта задания

Приложение з

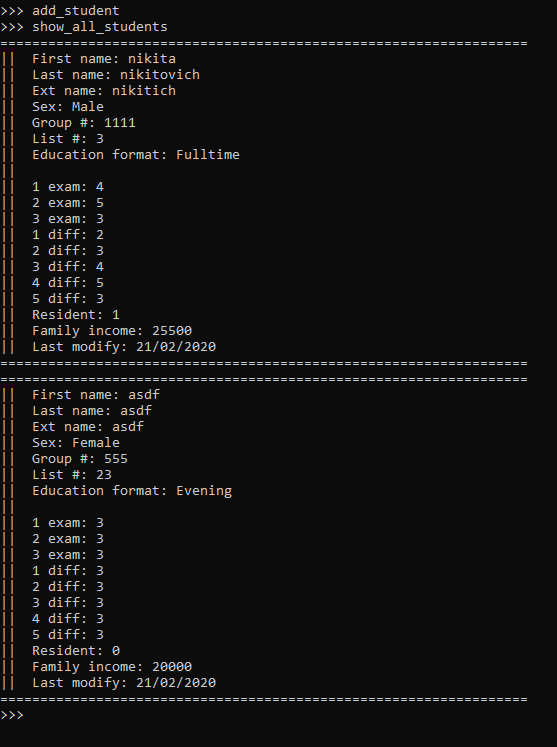
изображение 2



Изображение 2 – демонстрация работы второго пункта задания

Приложение И

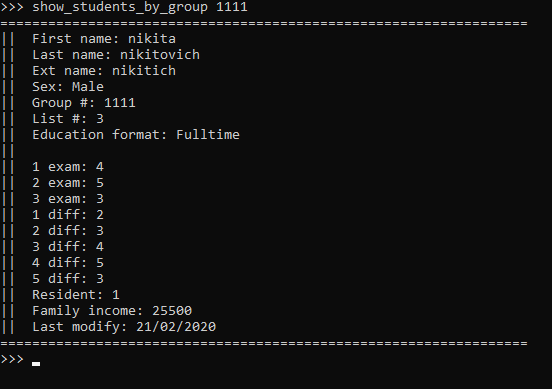
изображение 3



Изображение 3 – демонстрация работы третьего пункта задания

Приложение К

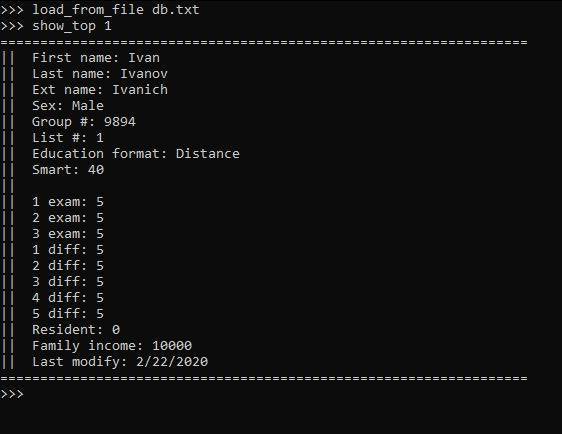
изображение 4



Изображение 4 – демонстрация работы четвертого пункта задания

Приложение Л

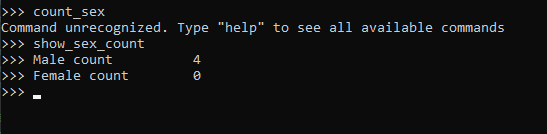
изображение 5



Изображение 5 – демонстрация работы пятого пункта задания

Приложение м

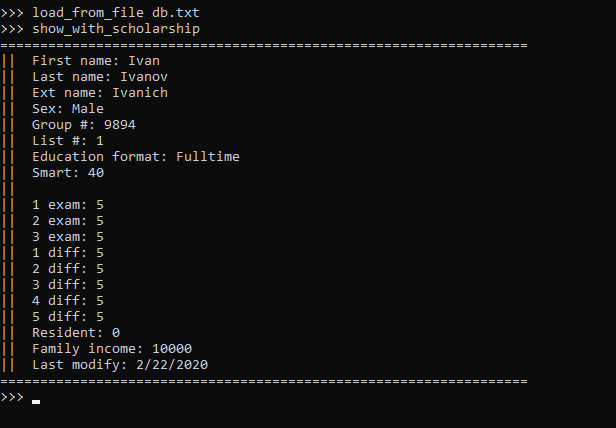
изображение 6



Изображение 6 – демонстрация работы шестого пункта задания

Приложение н

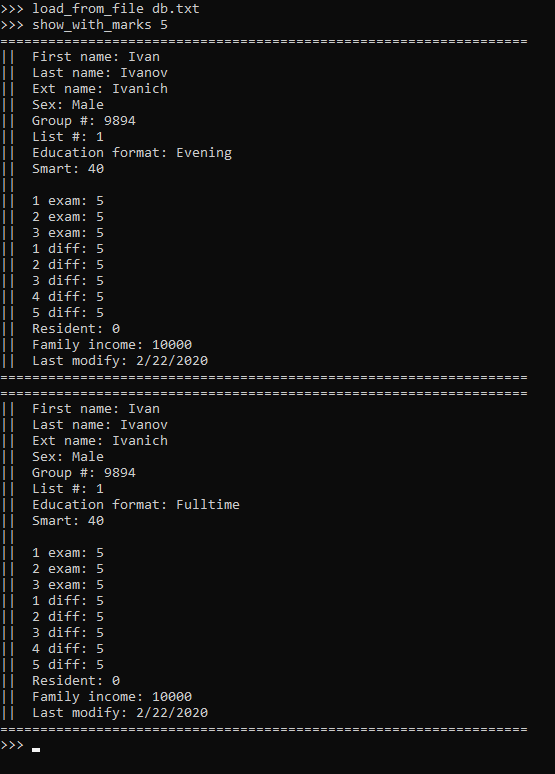
изображение 7



Изображение 7 – демонстрация работы седьмого пункта задания

Приложение о

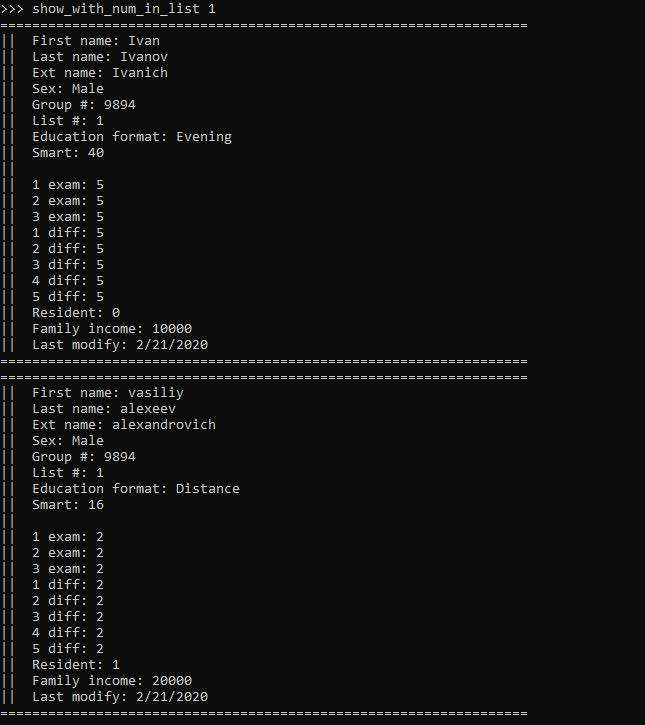
изображение 8



Изображение 8 – демонстрация работы восьмого пункта задания

Приложение п

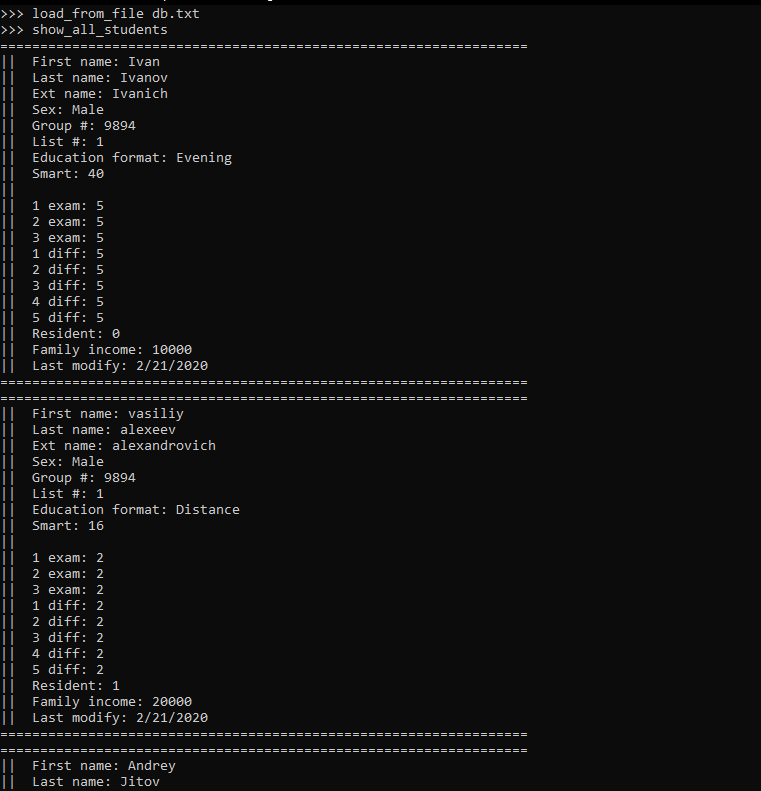
изображение 9



Изображение 9 – демонстрация девятого пункта задания

Приложение р

изображение 10



Изображение 10 – демонстрация работы десятого пункта задания