Пермский филиал федерального государственного автономного  
образовательного учреждения высшего образования  
«Национальный исследовательский университет  
«Высшая школа экономики»

*Факультет социально-экономических и компьютерных наук*

Мальшаков Александр Аркадьевич

**Самостоятельная работа**

студента образовательной программы «Разработка информационных систем для бизнеса» по направлению подготовки 38.03.05-Бизнес - информатика

Руководитель

Старший преподаватель кафедры ИТБ \_\_\_\_\_\_\_\_\_\_\_\_

Дацун Н. Н.

Пермь, 2025 год

# Оглавление

[Оглавление 2](#_Toc191323245)

[Запись 4](#_Toc191323246)

[Исходная табличка 5](#_Toc191323247)

[Массив 6](#_Toc191323248)

[Функция ввода данных 6](#_Toc191323249)

[С консоли 7](#_Toc191323250)

[С файла 7](#_Toc191323251)

[Случайно 8](#_Toc191323252)

[Вывод элементов массива 8](#_Toc191323253)

[Сортировка массива 9](#_Toc191323254)

[Функция поиска элементов по значениям ключевых атрибутов 10](#_Toc191323255)

[Бинарный поиск 10](#_Toc191323256)

[Функция редактирования записи в массиве 12](#_Toc191323257)

[Функция удаления элемента из массива 13](#_Toc191323258)

[Бинарное дерево 15](#_Toc191323259)

[Функция добавления записи в дерево 15](#_Toc191323260)

[Функция вывода данных 17](#_Toc191323261)

[Функция поиска элементов по значениям ключевых атрибутов 19](#_Toc191323262)

[Функция удаления записи из дерева 20](#_Toc191323263)

[Линейный список 24](#_Toc191323264)

[Функция добавление с упорядочиванием 24](#_Toc191323265)

[Просмотр записей в порядке ввода 25](#_Toc191323266)

[Просмотр записей в отсортированном порядке 26](#_Toc191323267)

[Поиск записи по значениям ключевых атрибутов 27](#_Toc191323268)

[Удаление записи 28](#_Toc191323269)

[Дополнительно 32](#_Toc191323270)

[Таблица 32](#_Toc191323271)

[Хеш Таблица 33](#_Toc191323272)

[Интерфейс 34](#_Toc191323273)

[Самооценка 36](#_Toc191323274)

[Библиографический список 37](#_Toc191323275)

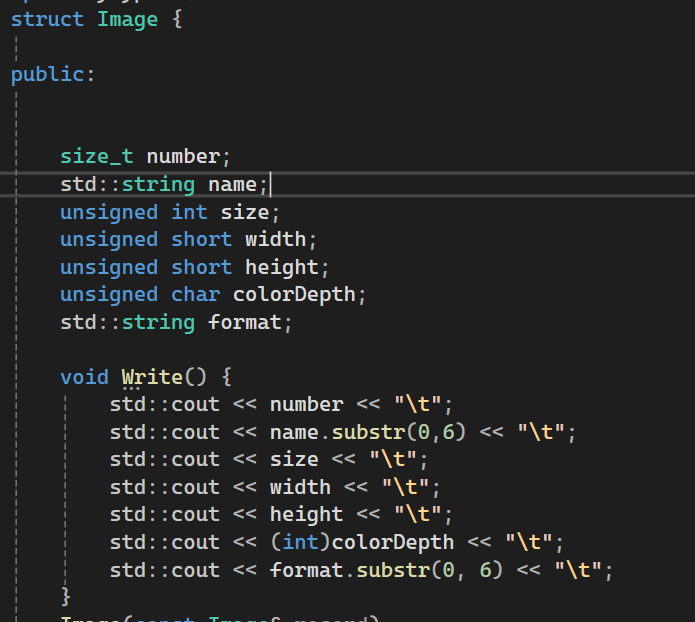
[Приложение А 38](#_Toc191323276)

[Приложение Б 39](#_Toc191323277)

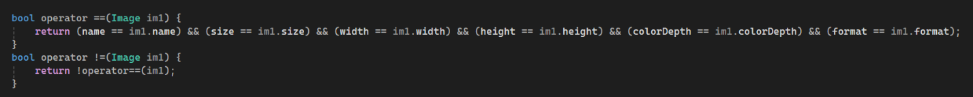
# Запись

Запись представляет собой структуру, хранящую номер, имя, размер в килобайтах, ширину, высоту, глубину цвета и формат картинки в базе данных. Для размера выбран тип unsigned int, так с помощью него можно задать картинке размер в 4 терабайта, изображений большего размера в мире не очень много. Для высоты и ширины выбран тип unsigned short так как картинки размером примерно 65тыс или больше пикселей очень-очень редки. Для глубины цвета использован один байт так как при глубине цвета больше 32 человеческий глаз перестает замечать разницу.

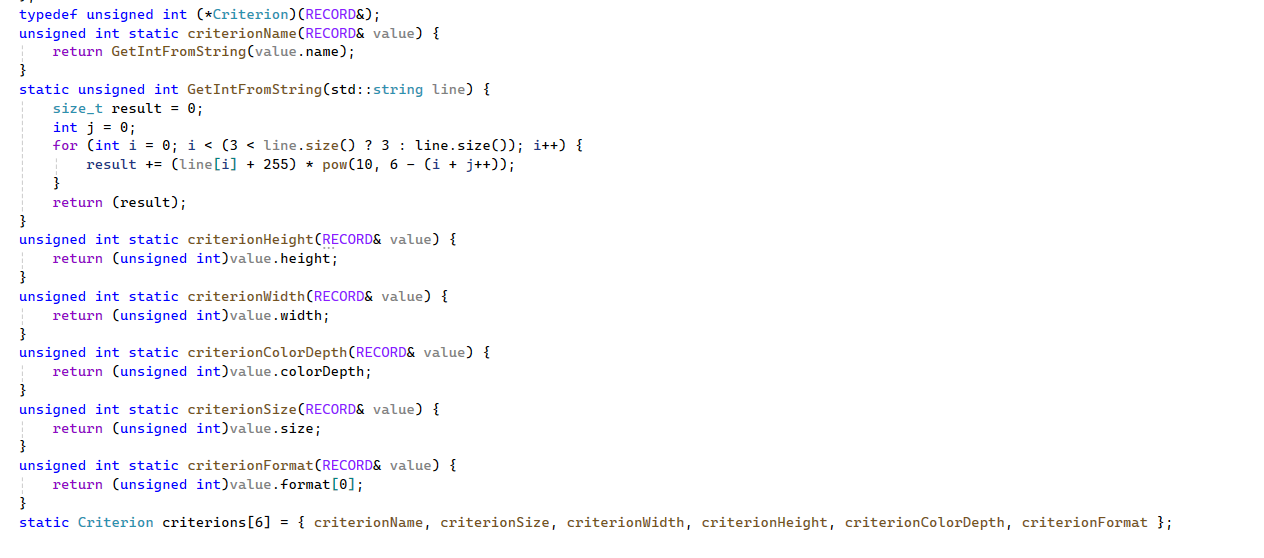
В записи есть также функция вывода, длинные названия сокращаются до 6 символов для красивого вывода.



Функция сравнения записи сравнивает все ее элементы.



Для того чтобы из записи получать ключи были написаны 6 функций. Все они позволяют сравнивать между собой ключи записи возвращая целое без знакового число, так как это самое большое поле в записи, и к нему все остальные будут приводиться без потери данных.



Ключ имени преобразуется в unsigned int следующим образом:

Суммируются первые 3 кода букв умноженные на 10 в степени их позиции, так сортировка будет лексикографической, коллизии могу происходить, но это достаточно редко.

## Исходная табличка

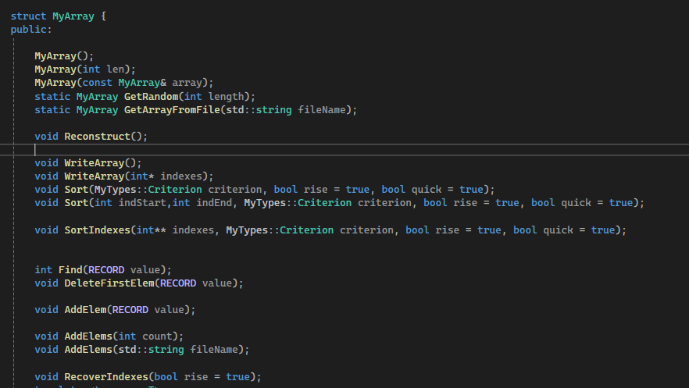
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Номер** | **Имя** | **Размер (кб)** | **Ширина** | **Высота** | **Глубина цвета** | **Формат** |
| 1 | Кот | 451 | 1262 | 713 | 8 | png |
| 2 | Собака на траве | 317 | 1400 | 1050 | 24 | jpg |
| 3 | Луна | 26 | 736 | 615 | 24 | jpg |
| 4 | Эйфелева башня | 428 | 3000 | 1688 | 32 | webp |
| 5 | Кремль | 165 | 1280 | 853 | 24 | jpg |
| 6 | Бабочка | 437 | 823 | 630 | 32 | jpg |
| 7 | Клубок | 430 | 820 | 635 | 32 | jpg |
| 8 | арахис | 45 | 100 | 100 | 32 | webp |
| 9 | Человек | 470 | 200 | 1630 | 32 | jpg |
| 10 | Шелёпа | 900 | 1280 | 1000 | 16 | png |
| 11 | Соболь | 300 | 1800 | 1350 | 8 | jpg |
| 12 | Карандаш | 200 | 3200 | 3223 | 8 | webp |
| 13 | Тополь | 500 | 456 | 355 | 16 | webp |

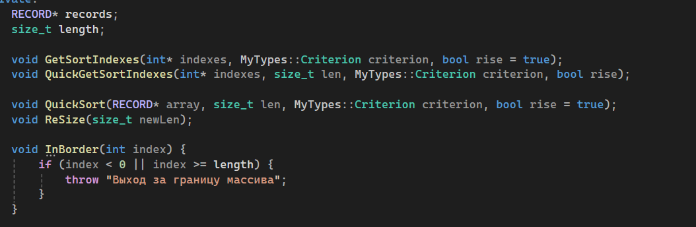
Ключи – имя и высота картинки.

# Массив

Массив хранит саму таблицу, список и дерево хранят указатели на массив так как это удобнее для вывода и проектирования функций, а также это не занимает больше памяти чем хранить индексы массива. Минус в том, что нужно заново создавать деревья и списки при изменении размера массива, но это проблема бы решилась если бы использовались умные указатели.

Массив представляет собой структуру, которая хранит указатель на массив записей, а также их количество и методы для работы с ними.

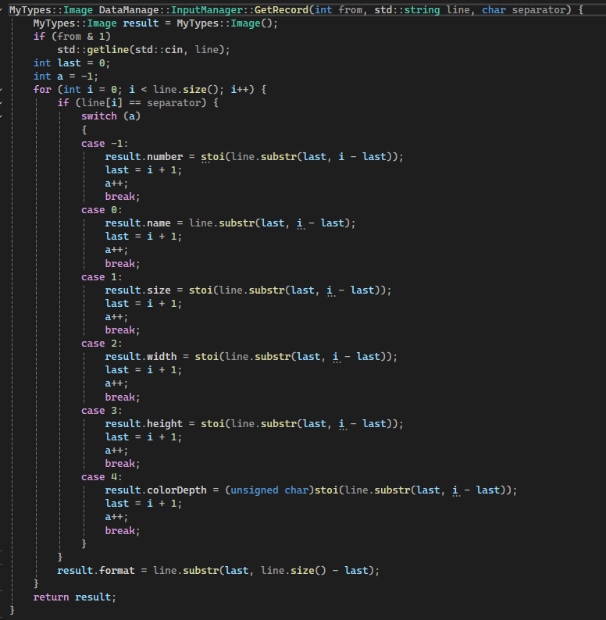




## Функция ввода данных

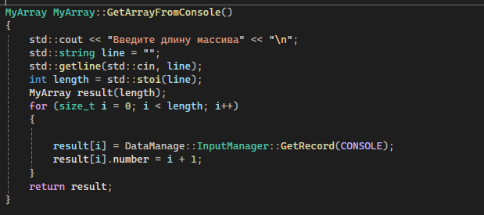
Для ввода массива были спроектированы 3 функции ввода, с консоли, с файла, с помощью генератора псевдослучайных чисел.

Для преобразования строки в запись была спроектирована функция.



### С консоли

Функция ввода массива с консоли.

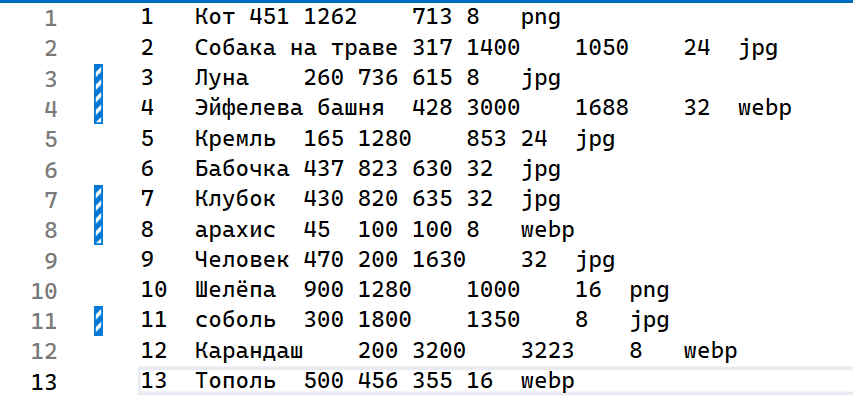


### С файла

Функция ввода массива с файла.

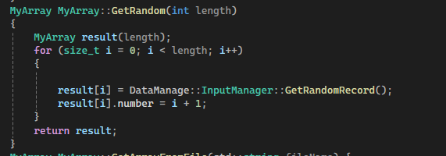


Сам файл, поля записи разделены символом “\t“.



### Случайно

Функция получения массива псевдо случайно.



Функция получения случайной записи.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

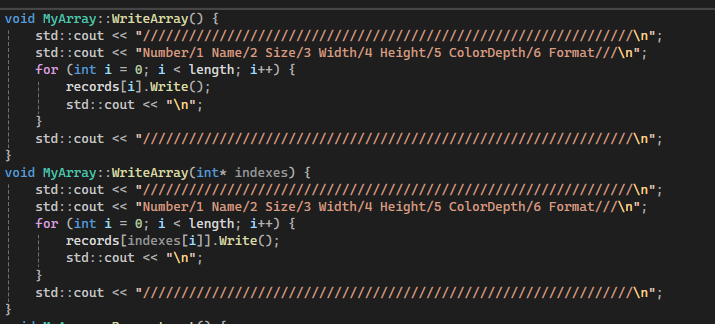
Функция получения случайной строки

Изображение выглядит как текст, снимок экрана, Шрифт

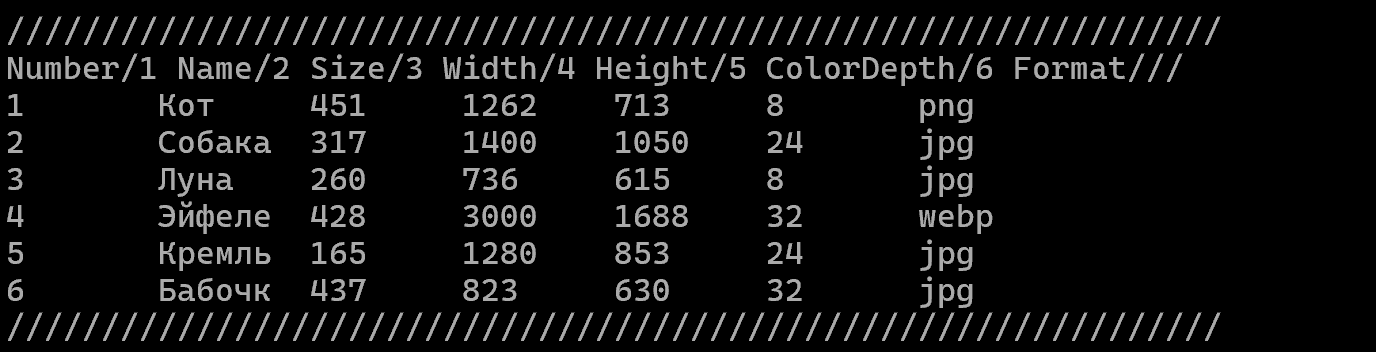
Контент, сгенерированный ИИ, может содержать ошибки.

## Вывод элементов массива

Для вывода массива были спроектированы 2 функции, одна выводит массив просто как таблицу, другая выводит массив по массив индексов.



Пример работы функции, выводящей просто массив.



Пример работы функции, выводящей по индексам (отсортировано по именам в обратном порядке).

Изображение выглядит как текст, снимок экрана, Шрифт, черный

Контент, сгенерированный ИИ, может содержать ошибки.

## Сортировка массива

Были созданы 2 метода, для быстрой сортировки и сортировки пузырьком. Они оба принимают пустой указатель на индексы и сортируют его.

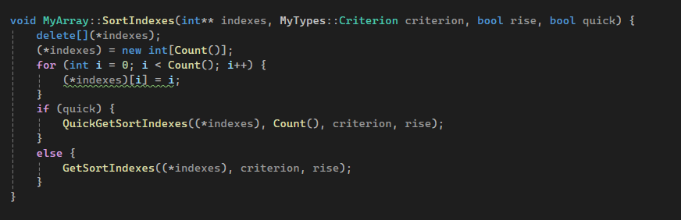


Сортировка пузырьком.

Изображение выглядит как текст, программное обеспечение, Шрифт, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Для создания указателя на индексы и передачи его в эти функции есть дополнительная функция.



Вывод массива по отсортированным индексам (сортировка оп высоте).

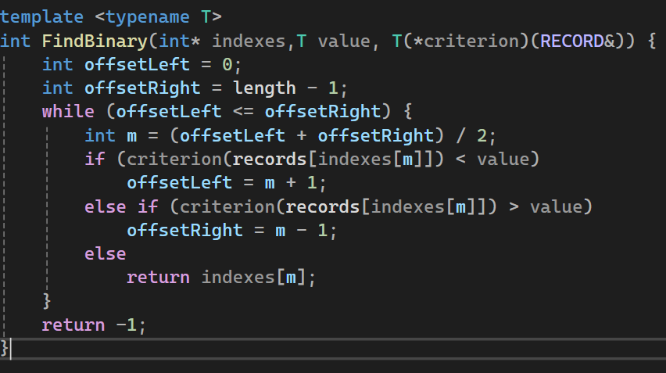
Изображение выглядит как текст, снимок экрана, Шрифт, черный

Контент, сгенерированный ИИ, может содержать ошибки.

## Функция поиска элементов по значениям ключевых атрибутов

### Бинарный поиск

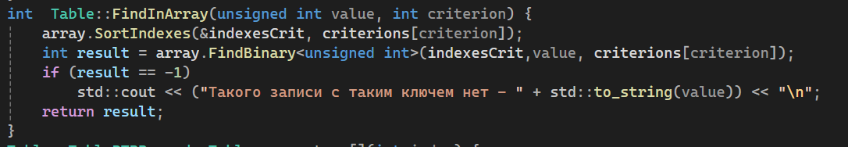
Для бинарного поиска спроектирована функция ниже итерационная и рекурсивная. Она принимает указатель на функцию для того, чтобы из записи получать ключ, по которому она будет искать, а также отсортированные индексы.



**Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.**

Функция, которая находит элемент в массиве, ниже. Так как в записи все ключи приводятся к типу unsigned int, то она передает в функцию значения типа unsigned int.



Ниже пример работы.

Изображение выглядит как текст, Шрифт, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

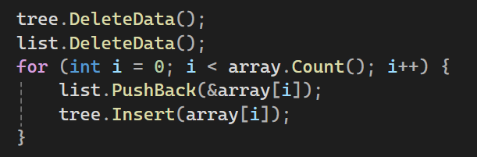
Изображение выглядит как текст, снимок экрана, Шрифт, черный

Контент, сгенерированный ИИ, может содержать ошибки.

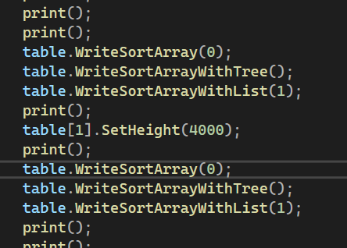


## Функция редактирования записи в массиве

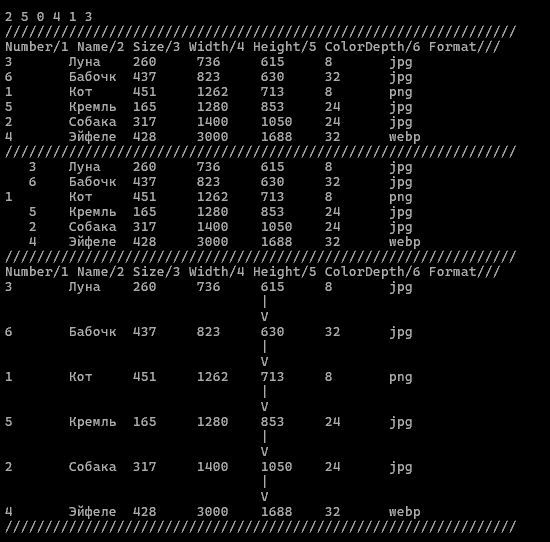
При редактировании нужно изменять индексы не только массива, а также дерева и списка. При редактировании записи будут заново создаваться списки и деревья.



Код.



Массив, дерево и список до изменения значения высоты у 2 записи.



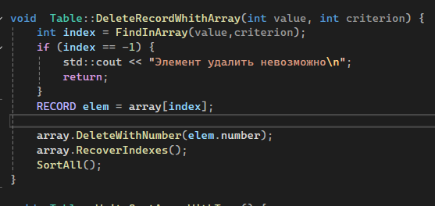
После изменения 2 записи.

Изображение выглядит как снимок экрана, текст

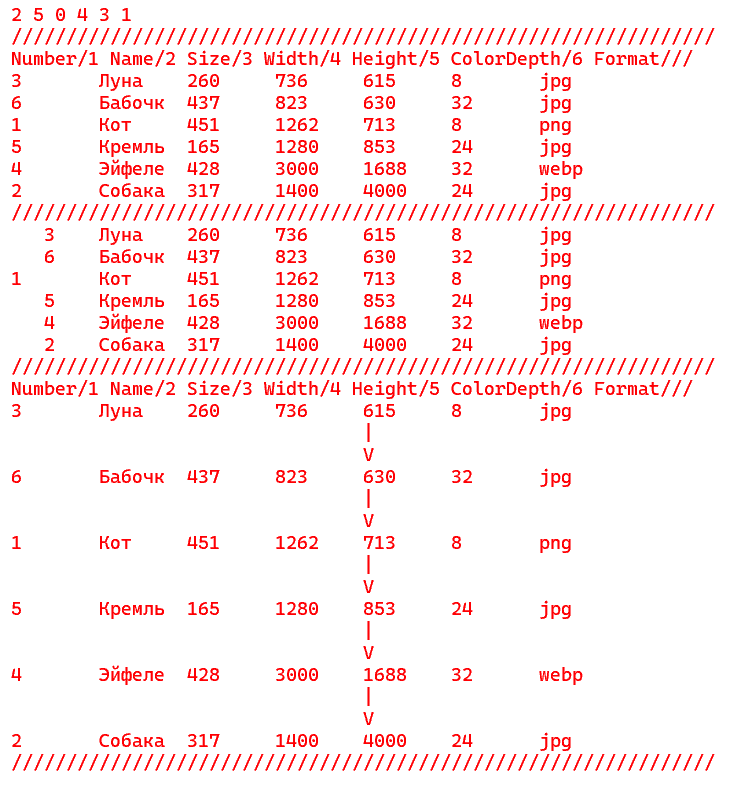
Контент, сгенерированный ИИ, может содержать ошибки.

## Функция удаления элемента из массива

Для удаления также надо заново создавать дерево и список, так как массив копирует все элементы в новую память и старые указатели начинают указывать на мусор. Функция удаления находит элемент в массиве с помощью предыдущей функции, потом удаляет его по индексу, восстанавливает номера записей и заново сортирует дерево и список.



До удаления элемента из массива с 713 высотой.



После

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

# Бинарное дерево

Дерево представляет собой структуру, хранящую корень – изначальный узел, а также функции для работы с ним. Для дерева выбран тип B-Tree для реализации балансировки. У B дерева есть порядок для него использовался шаблон, который позволяет задать переменную как константу при указании типа. За основу были взяты функции из статьи на хабре[1], но в них было несколько ошибок, которые пришлось исправлять.

Узел хранит массив детей, а также массив указателей на записи.



## Функция добавления записи в дерево

Принимает ссылку на запись, ищет лист если в листе есть место ставит запись туда, если же нет, то в функции Restruct, отправляет элемент в узел выше и делит текущий узел на 2, и присваивает эти 2 узла верхнему.

Изображение выглядит как текст, линия, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

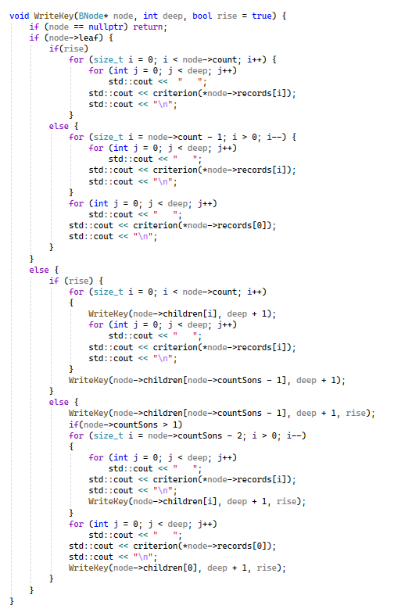
И так делает рекурсивно пока не останется переполненных узлов.

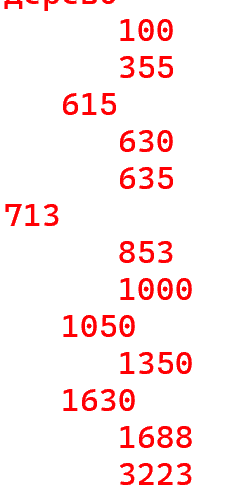
Изображение выглядит как текст, снимок экрана, документ, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

## Функция вывода данных

Функция для вывода только ключей из дерева.





Функция для вывода записей из дерева.

Изображение выглядит как текст, снимок экрана, Шрифт, документ

Контент, сгенерированный ИИ, может содержать ошибки.

Изображение выглядит как текст, Шрифт, число, снимок экрана

Контент, сгенерированный ИИ, может содержать ошибки.

## Функция поиска элементов по значениям ключевых атрибутов

Итерационная версия поисковой функции.

Изображение выглядит как текст, снимок экрана, программное обеспечение, Мультимедийное программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Рекурсивная

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

**Пример**

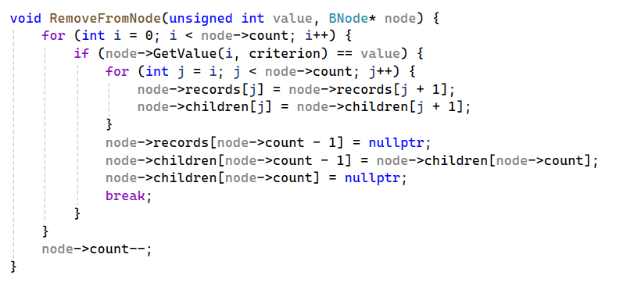
****

## Функция удаления записи из дерева

Для начала функция удаления проверяет нахождение элемента в дереве. После либо удаляет элемент из массива узла, либо из листа, либо из произвольного узла.



Метод удаления из массива узла.

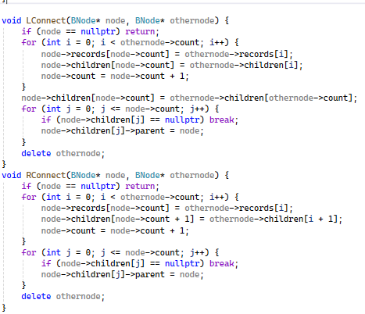


Метод удаления из листа.

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Методы левого и правого присоединения



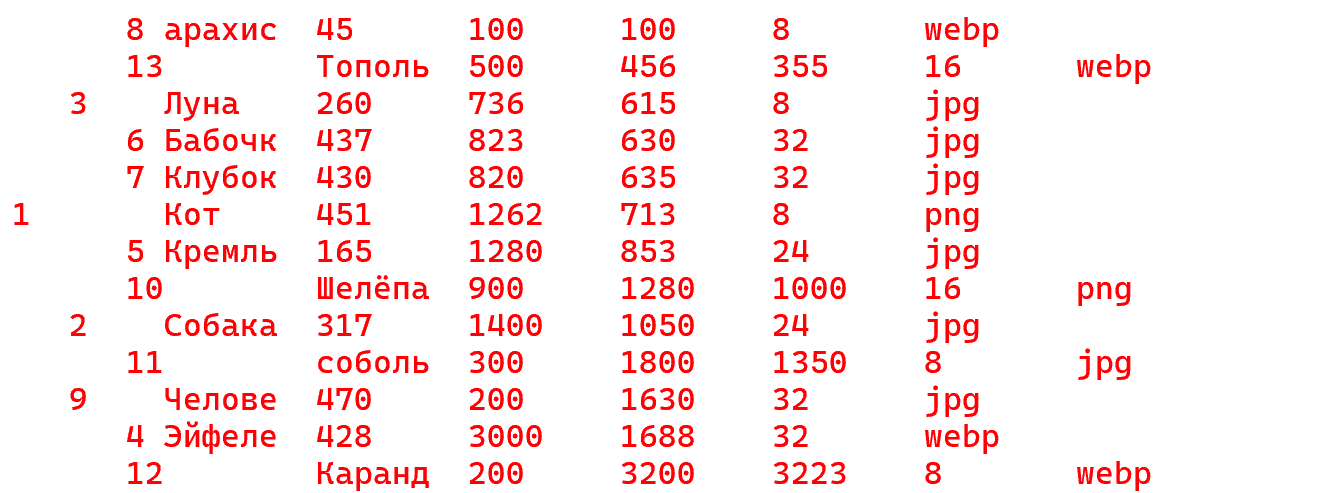
И метод починки узла.

Изображение выглядит как текст, снимок экрана, документ, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**Пример работы**

До удаления элемента с высотой 355



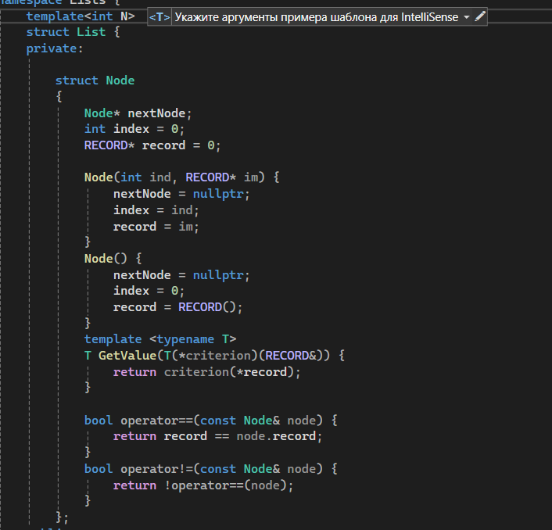
После удаления элемента с высотой 355

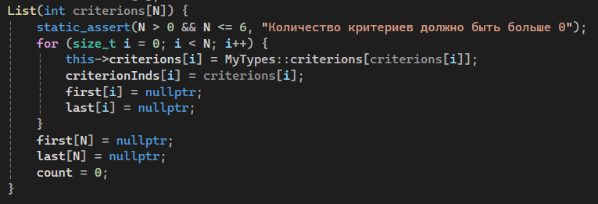
Изображение выглядит как текст, Шрифт, снимок экрана, число

Контент, сгенерированный ИИ, может содержать ошибки.

# Линейный список

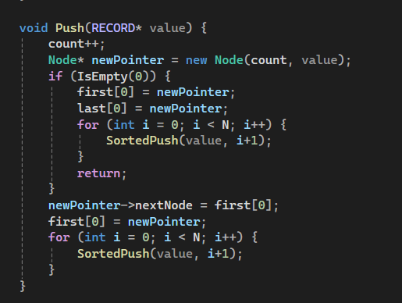
Список — это структура, хранящая массив из N указателей на первые узлы и на последние. Каждый указатель отвечает за сортируемый ключ, это удобнее чем если бы каждый узел указывал бы на 3 потомков, а памяти занимает столько же. Ключи передаются при создании списка, количество ключей определяется шаблоном.





## Функция добавление с упорядочиванием

Для начала функция принимает ссылку на запись, и либо вставляет в начало списка, либо в конец (функции Push и PushBack).

 Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

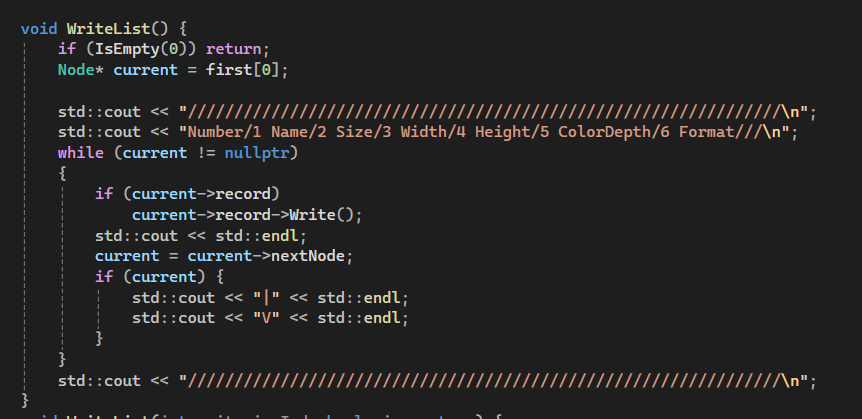
Дальше эти функции передают элемент в сортировочные функции для сортировки по ключам.



Так получаются N + 1 указатель, каждый отсортированный по своему ключу.

## Просмотр записей в порядке ввода

Нулевой указатель всегда указывает на элементы в порядке вставки. Ниже функция для вывода.

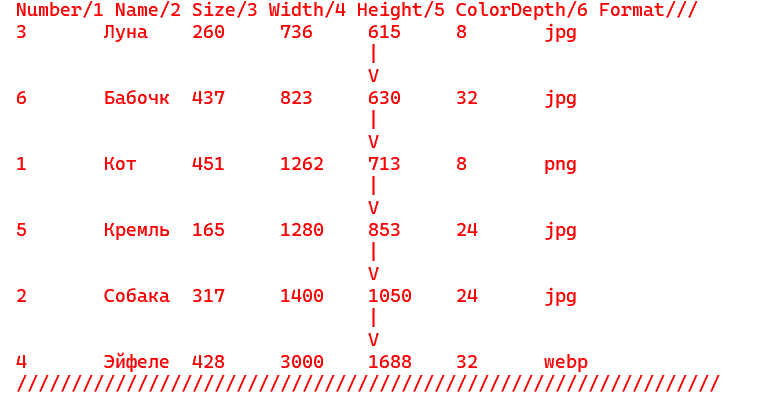


## Просмотр записей в отсортированном порядке

Для сортированного вывода нужно также вывести все элементы, но теперь уже не от 0-го указателя, для вывода по убыванию нужна функция переворачивания списка.



Список выводит записи со стрелочками, которые показывают на отсортированные ключи.



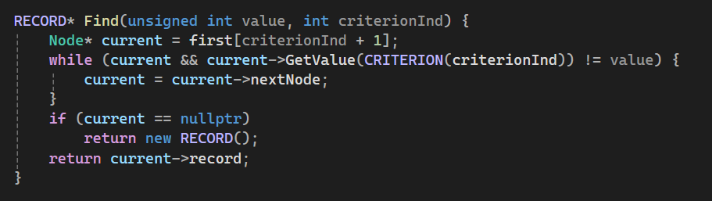
Функция переворачивания списка, итерационная и рекурсивная.

Изображение выглядит как текст, снимок экрана, Шрифт

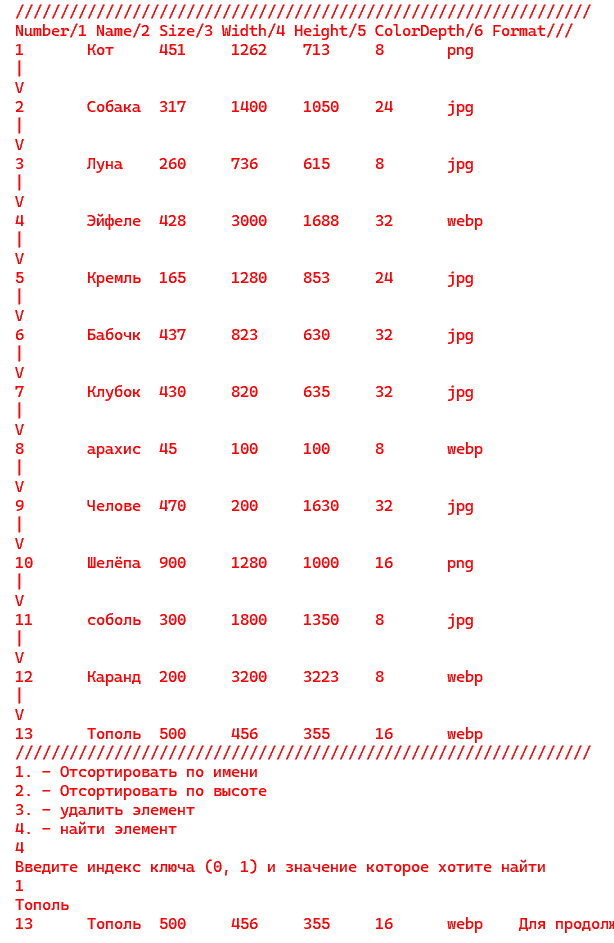
Контент, сгенерированный ИИ, может содержать ошибки. Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

## Поиск записи по значениям ключевых атрибутов



**Пример**



## Удаление записи

Чтобы удалить запись для начала ее нудно найти, потом удалить последовательно из всех списков по ключам.

Изображение выглядит как текст, снимок экрана, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Функции удаления первого/последнего элемента.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Функция удаления узла из всех под списков.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

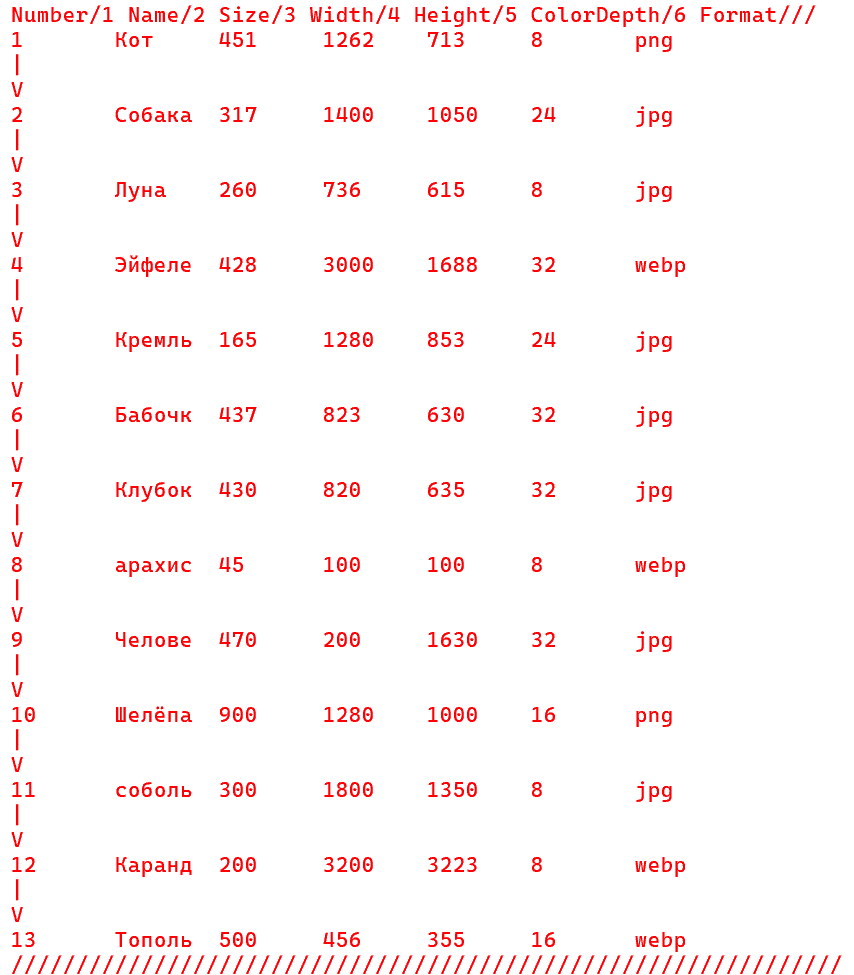
Функция удаления первого/последнего узла из подсписка.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

**Пример**

До удаления элемента с высотой 3223



После удаления элемента с высотой 3223

Изображение выглядит как текст, снимок экрана, Шрифт, меню

Контент, сгенерированный ИИ, может содержать ошибки.

# Дополнительно

## Таблица

Для объединения массива, дерева и списка сделана структура таблицы.



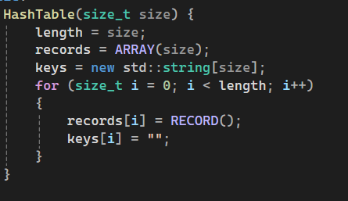
Через нее идет работа с 3 структурами.

В ней также есть структура табличного указателя на запись для того, чтобы, когда изменяли значение в записи, происходила сортировка всех структур.

Функции таблицы — это комбинации функций структур.

## Хеш Таблица

Хеш таблица сделана и работает, но ее негде применить.



Изображение выглядит как текст, снимок экрана, Шрифт, программное обеспечение

Контент, сгенерированный ИИ, может содержать ошибки.

Для того чтобы положить запись она высчитывает хэш строки, если есть колизия идет на следующую ячейку, если она так прошлась по всей таблице и не нашла куда вставить, то функция ничего не делает.

Изображение выглядит как текст, снимок экрана, программное обеспечение, дисплей

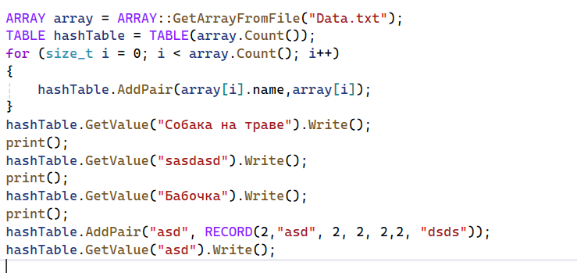
Контент, сгенерированный ИИ, может содержать ошибки.

Аналогичная функция получения только после она возвращает значение если оно было найдено, если же нет то возвращает пустую запись.

Изображение выглядит как текст, снимок экрана, Шрифт

Контент, сгенерированный ИИ, может содержать ошибки.

Пример работы, ниже код для теста.



Как видно она находит элемент по названию с помощью хэша.

Изображение выглядит как текст, Шрифт, красный

Контент, сгенерированный ИИ, может содержать ошибки.

## Интерфейс

Интерфейс позволяет просматривать таблицу через массив, список, дерево по 2 критериям – высоте и имени. Также из каждой структуры можно удалить или найти элементы.

Блок для массива



Блок для дерева

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

Блок для списка

Изображение выглядит как текст, снимок экрана, Шрифт, число

Контент, сгенерированный ИИ, может содержать ошибки.

# Самооценка

Сделаны все функции, которые были в задании, +ключи есть возможность поменять 2 действиями в коде программе + балансировка + хеш-таблица + некоторые функции рекурсивно и итерационно + интерфейс – мало проверок на правильный ввод в функциях и в интерфейсе

9

# Библиографический список

[1] Хабр “Сбалансированное дерево поиска B-Tree” [Электронный ресурс] -URL: https://habr.com/ru/articles/337594/(дата обращения: 20.02.2025)

# Приложение А

<https://github.com/Poruch/LabWorks/tree/main/IndividualWorkTOI> гит хаб

# Приложение Б

#include <string>

#include <iostream>

#define RECORD MyTypes::Image

namespace MyTypes {

    struct Image;

    static unsigned int criterionName(RECORD& value);

    struct Image {

    public:

        size\_t number;

        std::string name;

        unsigned int size;

        unsigned short width;

        unsigned short height;

        unsigned char colorDepth;

        std::string format;

        void Write() {

            std::cout << number << "\t";

            std::cout << name.substr(0, 6) << "\t";//"(" << std::to\_string(criterionName(\*this)) << ")" << "\t";

            std::cout << size << "\t";

            std::cout << width << "\t";

            std::cout << height << "\t";

            std::cout << (int)colorDepth << "\t";

            std::cout << format.substr(0, 6) << "\t";

        }

        Image(const Image& record)

            : number(record.number),

            name(record.name),

            size(record.size),

            width(record.width),

            height(record.height),

            colorDepth(record.colorDepth),

            format(record.format) { }

        Image(int number, const std::string& name, int size, int width, int height, int colorDepth, const std::string& format)

            : number(number),

            name(name),

            size(size),

            width(width),

            height(height),

            colorDepth(colorDepth),

            format(format) {    }

        Image(): number(0), name(""), size(0), width(0), height(0), colorDepth(0), format("")

        {

        }

        bool operator ==(Image im1) {

            return (name == im1.name) && (size == im1.size) && (width == im1.width) && (height == im1.height) && (colorDepth == im1.colorDepth) && (format == im1.format);

        }

        bool operator !=(Image im1) {

            return !operator==(im1);

        }

    };

    static unsigned int GetIntFromString(std::string line) {

        size\_t result = 0;

        int j = 0;

        for (int i = 0; i < (3 < line.size() ? 3 : line.size()); i++) {

            result += (line[i] + 255) \* pow(10, 6 - (i + j++));

        }

        return (result);

    }

    typedef unsigned int (\*Criterion)(RECORD&);

    unsigned int static criterionName(RECORD& value) {

        return GetIntFromString(value.name);

    }

    unsigned int static criterionHeight(RECORD& value) {

        return (unsigned int)value.height;

    }

    unsigned int static criterionWidth(RECORD& value) {

        return (unsigned int)value.width;

    }

    unsigned int static criterionColorDepth(RECORD& value) {

        return (unsigned int)value.colorDepth;

    }

    unsigned int static criterionSize(RECORD& value) {

        return (unsigned int)value.size;

    }

    unsigned int static criterionFormat(RECORD& value) {

        return (unsigned int)value.format[0];

    }

    static Criterion criterions[6] = { criterionName, criterionSize, criterionWidth, criterionHeight, criterionColorDepth, criterionFormat };

};

namespace DataManage {

    class InputManager {

    public:

        static RECORD GetRecord(int from, std::string line = "", char separator = ' ');

        static RECORD GetRandomRecord();

        static std::string randomStrGen(int length) {

            static std::string charset = "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890";

            std::string result;

            result.resize(length);

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

                result[i] = charset[rand() % charset.length()];

            return result;

        }

    };

}

MyTypes::Image DataManage::InputManager::GetRecord(int from, std::string line, char separator) {

    MyTypes::Image result = MyTypes::Image();

    if (from & 1)

        std::getline(std::cin, line);

    int last = 0;

    int a = -1;

    for (int i = 0; i < line.size(); i++) {

        if (line[i] == separator) {

            switch (a)

            {

            case -1:

                result.number = stoi(line.substr(last, i - last));

                last = i + 1;

                a++;

                break;

            case 0:

                result.name = line.substr(last, i - last);

                last = i + 1;

                a++;

                break;

            case 1:

                result.size = stoi(line.substr(last, i - last));

                last = i + 1;

                a++;

                break;

            case 2:

                result.width = stoi(line.substr(last, i - last));

                last = i + 1;

                a++;

                break;

            case 3:

                result.height = stoi(line.substr(last, i - last));

                last = i + 1;

                a++;

                break;

            case 4:

                result.colorDepth = (unsigned char)stoi(line.substr(last, i - last));

                last = i + 1;

                a++;

                break;

            }

        }

        result.format = line.substr(last, line.size() - last);

    }

    return result;

}

MyTypes::Image DataManage::InputManager::GetRandomRecord() {

    MyTypes::Image result = MyTypes::Image();

    result.name = randomStrGen(5);

    result.size = rand() % (1000 - 10 + 1) + 10;

    result.width = rand() % (5000 - 10 + 1) + 10;

    result.height = rand() % (5000 - 10 + 1) + 10;

    result.colorDepth =  pow(2, rand() % (8 - 1 + 1) + 1);

    result.format = randomStrGen(3);

    return result;

}

#define ARRAY Arrays::MyArray

namespace Arrays {

    struct MyArray {

    public:

        MyArray();

        MyArray(size\_t len);

        MyArray(const MyArray& array);

        static MyArray GetRandom(size\_t length);

        static MyArray GetArrayFromFile(std::string fileName);

        static MyArray GetArrayFromConsole();

        void Reconstruct();

        void WriteArray();

        void WriteArray(int\* indexes);

        void Sort(MyTypes::Criterion criterion, bool rise = true, bool quick = true);

        void Sort(size\_t indStart, size\_t indEnd, MyTypes::Criterion criterion, bool rise = true, bool quick = true);

        void SortIndexes(int\*\* indexes, MyTypes::Criterion criterion, bool rise = true, bool quick = true);

        int Find(RECORD value);

        void DeleteFirstElem(RECORD value);

        void AddElem(RECORD value);

        void AddElems(int count);

        void AddElems(std::string fileName);

        void RecoverIndexes(bool rise = true);

        template <typename T>

        int FindAll(int\*\* indexes, int\* lenIndexes, T value, T (\*criterion)(RECORD&)) {

            delete[](\*indexes);

            int count = 0;

            (\*indexes) = new int[length];

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

                if (criterion(records[i]) == value)

                    (\*indexes)[count++] = i;

            }

            if (count == 0)

                return -1;

            int\* result = new int[count];

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

                result[i] = (\*indexes)[i];

            }

            delete[](\*indexes);

            (\*indexes) = result;

            \*lenIndexes = count;

            return 1;

        }

        template <typename T>

        int FindBinary(int\* indexes,T value, T(\*criterion)(RECORD&)) {

            int offsetLeft = 0;

            int offsetRight = length - 1;

            while (offsetLeft <= offsetRight) {

                int m = (offsetLeft + offsetRight) / 2;

                if (criterion(records[indexes[m]]) < value)

                    offsetLeft = m + 1;

                else if (criterion(records[indexes[m]]) > value)

                    offsetRight = m - 1;

                else

                    return indexes[m];

            }

            return -1;

        }

        template <typename T>

        int FindBinary(int\* indexes,T value, T(\*criterion)(RECORD&), size\_t offsetLeft, size\_t offsetRight) {

            int m = (offsetLeft + offsetRight) / 2;

            if (offsetLeft > offsetRight)

                return -1;

            if (criterion(records[indexes[m]]) < value)

                return FindBinary(value, criterion, m + 1, offsetRight);

            else if (criterion(records[indexes[m]]) > value)

                return FindBinary(value, criterion, offsetLeft, m - 1);

            else

                return indexes[m];

        }

        template <typename T>

        int Find(T value, T(\*criterion)(RECORD&)) {

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

                if (criterion(records[i]) == value)

                    return i;

            return -1;

        }

        template <typename T>

        void DeleteElems(T value, T(\*criterion)(RECORD&)) {

            int\* indexes = 0;

            int lenIndexes = 0;

            if (!(FindAll<T>(&indexes, &lenIndexes, value, criterion) + 1))

                return;

            int count = 0;

            RECORD\* result = new RECORD[length - lenIndexes];

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

                if (i == indexes[count]) {

                    count++;

                }

                else {

                    result[i - count] = records[i];

                }

            }

            delete[] records;

            delete[] indexes;

            records = result;

            length -= lenIndexes;

            RecoverIndexes();

        }

        template <typename T>

        void DeleteFirstElem(T value, T(\*criterion)(RECORD&)) {

            int ind = Find<T>(value, criterion);

            if (ind == -1)

                return;

            RECORD\* result = new RECORD[length - 1];

            int j = 0;

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

            {

                result[j++] = records[i];

            }

            for (int i = ind + 1; i < length; i++)

            {

                result[j++] = records[i];

            }

            delete[] records;

            records = result;

            length -= 1;

            RecoverIndexes();

        }

        void DeleteWithNumber(size\_t value);

        RECORD& operator[](size\_t index) {

            InBorder(index);

            return records[index];

        }

        size\_t Count() const {

            return length;

        };

        RECORD\* GetData() {

            return records;

        }

        void DeleteData() {

            delete[] records;

            records = new RECORD[0];

            length = 0;

        }

    private:

        RECORD\* records;

        size\_t length;

        void GetSortIndexes(int\* indexes, MyTypes::Criterion criterion, bool rise = true);

        void QuickGetSortIndexes(int\* indexes, size\_t len, MyTypes::Criterion criterion, bool rise);

        void QuickSort(RECORD\* array, size\_t len, MyTypes::Criterion criterion, bool rise = true);

        void ReSize(size\_t newLen);

        void InBorder(int index) {

            if (index < 0 || index >= length) {

                throw "Выход за границу массива";

            }

        }

    };

}

#define TABLE HashTables::HashTable

namespace HashTables {

    struct HashTable {

    public:

        HashTable(size\_t size) {

            length = size;

            records = ARRAY(size);

            keys = new std::string[size];

            for (size\_t i = 0; i < length; i++)

            {

                records[i] = RECORD();

                keys[i] = "";

            }

        }

        RECORD GetValue(std::string key) {

            int hash = GetHash(key);

            int count = 1;

            while (keys[hash] != key && count != length) {

                hash++;

                count++;

                hash = hash % length;

            }

            if (count == length)

                return RECORD();

            return records[hash];

        }

        void AddPair(std::string key, RECORD value) {

            int hash = GetHash(key);

            int count = 1;

            while (keys[hash] != "" && count != length) {

                hash++;

                count++;

                hash = hash % length;

            }

            if (count == length)

                return;

            if (keys[hash] == key) {

                return;

            }

            records[hash] = value;

            keys[hash] = key;

        }

    private:

        int GetHash(std::string key) {

            int sum = 0;

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

                sum += abs(key[i]);

            }

            int hash = sum % length;

            return hash;

        }

        size\_t length;

        ARRAY records;

        std::string\* keys;

    };

}

#define TREE Trees::BTree

namespace Trees {

template<int N>

struct BTree {

private:

    struct BNode

    {

        RECORD\*\* records;

        BNode\* children[2 \* N + 1];

        BNode\* parent;

        int count;

        int countSons;

        bool leaf;

        BNode() : leaf(true) {

            records = new RECORD\*[2 \* N];

            for (int i = 0; i < 2 \* N; i++) records[i] = nullptr;

            for (int i = 0; i <= 2 \* N; i++) children[i] = nullptr;

            parent = nullptr;

            count = 0;

            countSons = 0;

        };

        BNode(bool isLeaf) : leaf(isLeaf) {

            records = new RECORD \* [2 \* N];

            for (int i = 0; i < 2 \* N; i++) records[i] = nullptr;

            for (int i = 0; i <= 2 \* N; i++) children[i] = nullptr;

            parent = nullptr;

            count = 0;

            countSons = 0;

        };

        unsigned int GetValue(int index, MyTypes::Criterion criterion) {

            return criterion(\*records[index]);

        }

        RECORD& operator[](int index) {

            return records[index];

        }

    };

    MyTypes::Criterion criterion;

    BNode\* root;

public:

    BTree(int criterionInd) {

        root = nullptr;

        this->criterion = MyTypes::criterions[criterionInd];

    }

    BTree(const BTree& tree) {

        root = tree.root;

    }

    BTree() {

        root = nullptr;

        this->criterion = MyTypes::criterionName;

    }

    void DeleteData() {

        if (root != nullptr)

            DeleteNode(root);

    }

    RECORD Find(unsigned int value) {

        return GetRecordWhithKey(value);

    }

    bool Contains(unsigned int value) {

        return FindKey(value,root);

    }

    void Insert(RECORD& value) {

        if (IsEmpty()) {

            BNode\* newRoot = new BNode();

            newRoot->records[0] = &value;

            newRoot->count++;

            root = newRoot;

        }

        else {

            BNode\* current = root;

            while(!current->leaf) {

                for (int i = 0; i < 2 \* N; i++) {

                    if (current->records[i] != nullptr) {

                        if (criterion(value) <= criterion((\*current->records[i]))) {

                            current = current->children[i];

                            break;

                        }

                        if (((i + 1) == current->count || current->records[i + 1] == nullptr) && (criterion(value) > criterion((\*current->records[i])))) {

                            current = current->children[i + 1];

                            break;

                        }

                    }

                    else break;

                }

            }

            InsertToNode(value, current);

            while (current->count == 2 \* N) {

                if (current == root) {

                    Restruct(current);

                    break;

                }

                else {

                    Restruct(current);

                    BNode\* oldCurrent = current;

                    current = current->parent;

                    delete oldCurrent;

                }

            }

        }

    }

    void Delete(unsigned int value) {

        BNode\* current = this->root;

        int position;

        int positionSon;

        int i;

        if (!FindKey(value)) {

            return;

        }

        else {

            for (i = 0; i < current->count; i++) {

                if (\*current->records[i] != RECORD()) {

                    if (value == current->GetValue(i, criterion)) {

                        position = i;

                        break;

                    }

                    else {

                        if ((value < current->GetValue(i, criterion))) {

                            current = current->children[i];

                            positionSon = i;

                            i = -1;

                        }

                        else {

                            if (i == (current->count - 1)) {

                                current = current->children[i + 1];

                                positionSon = i + 1;

                                i = -1;

                            }

                        }

                    }

                }

                else break;

            }

        }

        if (current->leaf) {

            if (current->count >= N)

                RemoveFromNode(value, current);

            else

                RemoveLeaf(value, current);

        }

        else

            Remove(value, current);

    }

    void WriteTree(bool rise = true) {

        WriteNode(root,0,rise);

    }

    void WriteTreeKeys(bool rise = true) {

        WriteKey(root, 0,rise);

    }

    bool IsEmpty() {

        return root == nullptr;

    }

private:

    void WriteNode(BNode\* node, int deep, bool rise = true) {

        if (node == nullptr) return;

        if (node->leaf) {

            if(rise)

                for (size\_t i = 0; i < node->count; i++)

                {

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

                        std::cout << "   ";

                    node->records[i]->Write();

                    std::cout << "\n";

                }

            else {

                for (size\_t i = node->count - 1; i > 0; i--) {

                    for (int j = 0; j < deep; j++)

                        std::cout << "   ";

                    node->records[i]->Write();

                    std::cout << "\n";

                }

                for (int j = 0; j < deep; j++)

                    std::cout << "   ";

                node->records[0]->Write();

                std::cout << "\n";

            }

        }

        else {

            if (rise) {

                for (size\_t i = 0; i < node->count; i++)

                {

                    WriteNode(node->children[i], deep + 1);

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

                        std::cout << "   ";

                    node->records[i]->Write();

                    std::cout << "\n";

                }

                WriteNode(node->children[node->countSons - 1], deep + 1);

            }

            else {

                WriteNode(node->children[node->countSons - 1], deep + 1, rise);

                if (node->countSons > 1)

                for (size\_t i = node->countSons - 2; i > 0; i--)

                {

                    for (int j = 0; j < deep; j++)

                        std::cout << "   ";

                    node->records[i]->Write();

                    std::cout << "\n";

                    WriteNode(node->children[i], deep + 1, rise);

                }

                for (int j = 0; j < deep; j++)

                    std::cout << "   ";

                node->records[0]->Write();

                std::cout << "\n";

                WriteNode(node->children[0], deep + 1, rise);

            }

        }

    }

    void WriteKey(BNode\* node, int deep, bool rise = true) {

        if (node == nullptr) return;

        if (node->leaf) {

            if(rise)

                for (size\_t i = 0; i < node->count; i++) {

                    for (int j = 0; j < deep; j++)

                        std::cout <<  "   ";

                    std::cout << criterion(\*node->records[i]);

                    std::cout << "\n";

                }

            else {

                for (size\_t i = node->count - 1; i > 0; i--) {

                    for (int j = 0; j < deep; j++)

                        std::cout << "   ";

                    std::cout << criterion(\*node->records[i]);

                    std::cout << "\n";

                }

                for (int j = 0; j < deep; j++)

                    std::cout << "   ";

                std::cout << criterion(\*node->records[0]);

                std::cout << "\n";

            }

        }

        else {

            if (rise) {

                for (size\_t i = 0; i < node->count; i++)

                {

                    WriteKey(node->children[i], deep + 1);

                    for (int j = 0; j < deep; j++)

                        std::cout << "   ";

                    std::cout << criterion(\*node->records[i]);

                    std::cout << "\n";

                }

                WriteKey(node->children[node->countSons - 1], deep + 1);

            }

            else {

                WriteKey(node->children[node->countSons - 1], deep + 1, rise);

                if(node->countSons > 1)

                for (size\_t i = node->countSons - 2; i > 0; i--)

                {

                    for (int j = 0; j < deep; j++)

                        std::cout << "   ";

                    std::cout << criterion(\*node->records[i]);

                    std::cout << "\n";

                    WriteKey(node->children[i], deep + 1, rise);

                }

                for (int j = 0; j < deep; j++)

                    std::cout << "   ";

                std::cout << criterion(\*node->records[0]);

                std::cout << "\n";

                WriteKey(node->children[0], deep + 1, rise);

            }

        }

    }

    void InsertToNode(RECORD& value, BNode\* node) {

        node->records[node->count] = &value;

        node->count = node->count + 1;

        SortNode(node);

    }

    void SortNode(BNode\* node) {

        for (int i = 0; i < node->count -1; i++) {

            for (int j = i + 1; j < node->count; j++) {

                if (criterion(\*node->records[i]) > criterion(\*node->records[j])) {

                    auto buff = node->records[i];

                    node->records[i] = node->records[j];

                    node->records[j] = buff;

                }

            }

        }

    }

    void Restruct(BNode\* node) {

        if (node->count < 2 \* N ) return;

        BNode\* child1 = new BNode();

        for (int j = 0; j < N - 1; j++)

            child1->records[j] = node->records[j];

        child1->count = N - 1;

        if (node->countSons != 0) {

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

                child1->children[i] = node->children[i];

                child1->children[i]->parent = child1;

            }

            child1->leaf = false;

            child1->countSons = N ;

        }

        else {

            child1->leaf = true;

            child1->countSons = 0;

        }

        BNode\* child2 = new BNode();

        for (int j = 0; j < N ; j++)

            child2->records[j] = node->records[j + N];

        child2->count = N;

        if (node->countSons != 0) {

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

                child2->children[i] = node->children[i + N];

                child2->children[i]->parent = child2;

            }

            child2->leaf = false;

            child2->countSons = N+1;

        }

        else {

            child2->leaf = true;

            child2->countSons = 0;

        }

        if (node->parent == nullptr) {

            node->records[0] = node->records[N - 1];

            node->children[0] = child1;

            node->children[1] = child2;

            for (int i = 2; i < node->countSons; i++) {

                node->children[i] = nullptr;

            }

            for (int i = 1; i < node->count; i++) {

                node->records[i] = nullptr;

            }

            node->leaf = false;

            node->count = 1;

            node->countSons = 2;

            child1->parent = node;

            child2->parent = node;

        }

        else {

            InsertToNode(\*node->records[N - 1], node->parent);

            for (int i = 0; i <= 2 \* N; i++) {

                if (node->parent->children[i] == node)

                    node->parent->children[i] = nullptr;

            }

            for (int i = 0; i <= 2 \* N; i++) {

                if (node->parent->children[i] == nullptr) {

                    for (int j = 2 \* N; j > i + 1; j--)

                        node->parent->children[j] = node->parent->children[j - 1];

                    node->parent->children[i + 1] = child2;

                    node->parent->children[i] = child1;

                    node->parent->countSons++;

                    break;

                }

            }

            child1->parent = node->parent;

            child2->parent = node->parent;

            node->parent->leaf = false;

        }

    }

    void DeleteNode(BNode\* node) {

        if (node != nullptr) {

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

                if (node->children[i] != nullptr) DeleteNode(node->children[i]);

                else {

                    delete node;

                    break;

                }

            }

        }

        root = nullptr;

    }

    bool FindKey(unsigned int value, BNode\* node) {

        if (node != nullptr) {

            if (!node->leaf) {

                int i;

                for (i = 0; i < 2 \* N ; i++) {

                    if (!(\*node->records[i] == RECORD())) {

                        if (value == node->GetValue(i, criterion)) return true;

                        if ((value < node->GetValue(i, criterion))) {

                            return FindKey(value, node->children[i]);

                            break;

                        }

                    }

                    else break;

                }

                return FindKey(value, node->children[i]);

            }

            else {

                for (int j = 0; j < node->count; j++)

                    if (value == node->GetValue(j, criterion)) return true;

                return false;

            }

        }

        else return false;

    }

    bool FindKey(unsigned int value) {

        BNode\* current = root;

        while (!current->leaf) {

            for (int i = 0; i < 2 \* N; i++) {

                if (\*current->records[i] != RECORD()) {

                    if (value == criterion(\*current->records[i])) {

                        return true;

                    }

                    if (value < criterion(\*current->records[i])) {

                        current = current->children[i];

                        break;

                    }

                    if (((i + 1) == current->count || \*current->records[i + 1] == RECORD()) && (value > criterion(\*current->records[i]))) {

                        current = current->children[i + 1];

                        break;

                    }

                }

                else

                    return false;

            }

        }

        bool result = false;

        for (int i = 0; i < current->count; i++) {

            result =  result || (current->GetValue(i,criterion) ==value);

        }

        return result;

    }

    RECORD GetRecordWhithKey(unsigned int value) {

        BNode\* current = root;

        while (!current->leaf) {

            for (int i = 0; i < 2 \* N; i++) {

                if (\*current->records[i] != RECORD()) {

                    if (value == criterion(\*current->records[i])) {

                        return \*current->records[i];

                    }

                    if (value < criterion(\*current->records[i])) {

                        current = current->children[i];

                        break;

                    }

                    if (((i + 1) == current->count || \*current->records[i + 1] == RECORD()) && (value > criterion(\*current->records[i]))) {

                        current = current->children[i + 1];

                        break;

                    }

                }

                else

                    return RECORD();

            }

        }

        RECORD result = RECORD();

        for (int i = 0; i < current->count; i++) {

            if (current->GetValue(i,criterion) == value)

                result = \*current->records[i];

        }

        return result;

    }

    RECORD GetRecordWhithKey(unsigned int value, BNode\* node) {

        if (node != nullptr) {

            if (!node->leaf) {

                int i;

                for (i = 0; i < 2 \* N; i++) {

                    if (!(\*node->records[i] == RECORD())) {

                        if (value == node->GetValue(i, criterion)) return \*node->records[i];

                        if ((value < node->GetValue(i, criterion))) {

                            return GetRecordWhithKey(value, node->children[i]);

                            break;

                        }

                    }

                    else break;

                }

                return GetRecordWhithKey(value, node->children[i]);

            }

            else {

                for (int j = 0; j < node->count; j++)

                    if (value == node->GetValue(j, criterion)) return \*node->records[j];

                return RECORD();

            }

        }

        else return RECORD();

    }

    void Remove(unsigned int value, BNode\* node) {

        BNode\* current = node;

        int position;

        for (int i = 0; i <= node->count - 1; i++) {

            if (value == node->GetValue(i, criterion)) {

                position = i;

                break;

            }

        }

        int positionSon;

        if (current->parent != nullptr) {

            for (int i = 0; i <= current->parent->count; i++) {

                if (current->children[i] == current) {

                    positionSon == i;

                    break;

                }

            }

        }

        current = current->children[position + 1];

        RECORD\* newkey = current->records[0];

        while (current->leaf == false)

            current = current->children[0];

        if (current->count > (N - 1)) {

            newkey = current->records[0];

            RemoveFromNode(criterion(\*newkey), current);

            node->records[position] = newkey;

        }

        else {

            current = node;

            current = current->children[position];

            newkey = current->records[current->count - 1];

            while (current->leaf == false) current = current->children[current->count];

            newkey = current->records[current->count - 1];

            node->records[position] = newkey;

            if (current->count > (N - 1))

                RemoveFromNode(criterion(\*newkey), current);

            else {

                RemoveLeaf(criterion(\*newkey), current);

            }

        }

    }

    void RemoveFromNode(unsigned int value, BNode\* node) {

        for (int i = 0; i < node->count; i++) {

            if (node->GetValue(i, criterion) == value) {

                for (int j = i; j < node->count; j++) {

                    node->records[j] = node->records[j + 1];

                    node->children[j] = node->children[j + 1];

                }

                node->records[node->count - 1] = nullptr;

                node->children[node->count - 1] = node->children[node->count];

                node->children[node->count] = nullptr;

                break;

            }

        }

        node->count--;

    }

    void RemoveLeaf(unsigned int value, BNode\* node) {

        if ((node == root) && (node->count == 1)) {

            RemoveFromNode(value, node);

            root->children[0] = nullptr;

            delete root;

            root = nullptr;

            return;

        }

        if (node == root) {

            RemoveFromNode(value, node);

            return;

        }

        if (node->count > (N - 1)) {

            RemoveFromNode(value, node);

            return;

        }

        BNode\* current = node;

        RECORD\* k1;

        RECORD\* k2;

        int position;

        int positionSon;

        int i;

        for (int i = 0; i <= node->count - 1; i++) {

            if (value == node->GetValue(i, criterion)) {

                position = i;

                break;

            }

        }

        BNode\* parent = current->parent;

        for (int j = 0; j <= parent->count; j++) {

            if (parent->children[j] == current) {

                positionSon = j;

                break;

            }

        }

        if (positionSon == 0) {

            if (parent->children[positionSon + 1]->count > (N - 1)) {

                k1 = parent->children[positionSon + 1]->records[0];

                k2 = parent->records[positionSon];

                InsertToNode(\*k2, current);

                RemoveFromNode(value, current);

                parent->records[positionSon] = k1;

                RemoveFromNode(criterion(\*k1), parent->children[positionSon + 1]);

            }

            else {

                RemoveFromNode(value, current);

                if (current->count <= (N - 2))

                    Repair(current);

            }

        }

        else {

            if (positionSon == parent->count) {

                if (parent->children[positionSon - 1]->count > (N - 1)) {

                    BNode\* temp = parent->children[positionSon - 1];

                    k1 = temp->records[temp->count - 1];

                    k2 = parent->records[positionSon - 1];

                    InsertToNode(\*k2, current);

                    RemoveFromNode(value, current);

                    parent->records[positionSon - 1] = k1;

                    RemoveFromNode(criterion(\*k1), temp);

                }

                else {

                    RemoveFromNode(value, current);

                    if (current->count <= (N - 2))

                        Repair(current);

                }

            }

            else {

                if (parent->children[positionSon + 1]->count > (N - 1)) {

                    k1 = parent->children[positionSon + 1]->records[0];

                    k2 = parent->records[positionSon];

                    InsertToNode(\*k2, current);

                    RemoveFromNode(value, current);

                    parent->records[positionSon] = k1;

                    RemoveFromNode(criterion(\*k1), parent->children[positionSon + 1]);

                }

                else {

                    if (parent->children[positionSon - 1]->count > (N - 1)) {

                        BNode\* temp = parent->children[positionSon - 1];

                        k1 = temp->records[temp->count - 1];

                        k2 = parent->records[positionSon - 1];

                        InsertToNode(\*k2, current);

                        RemoveFromNode(value, current);

                        parent->records[positionSon - 1] = k1;

                        RemoveFromNode(criterion(\*k1), temp);

                    }

                    else {

                        RemoveFromNode(value, current);

                        if (current->count <= (N - 2)) Repair(current);

                    }

                }

            }

        }

    }

    void LConnect(BNode\* node, BNode\* othernode) {

        if (node == nullptr) return;

        for (int i = 0; i < othernode->count; i++) {

            node->records[node->count] = othernode->records[i];

            node->children[node->count] = othernode->children[i];

            node->count = node->count + 1;

        }

        node->children[node->count] = othernode->children[othernode->count];

        for (int j = 0; j <= node->count; j++) {

            if (node->children[j] == nullptr) break;

            node->children[j]->parent = node;

        }

        delete othernode;

    }

    void RConnect(BNode\* node, BNode\* othernode) {

        if (node == nullptr) return;

        for (int i = 0; i < othernode->count; i++) {

            node->records[node->count] = othernode->records[i];

            node->children[node->count + 1] = othernode->children[i + 1];

            node->count = node->count + 1;

        }

        for (int j = 0; j <= node->count; j++) {

            if (node->children[j] == nullptr) break;

            node->children[j]->parent = node;

        }

        delete othernode;

    }

    void Repair(BNode\* node) {

        if ((node == root) && (node->count == 0)) {

            if (root->children[0] != nullptr) {

                root->children[0]->parent = nullptr;

                root = root->children[0];

            }

            else {

                delete root;

            }

            return;

        }

        BNode\* current = node;

        RECORD k1;

        RECORD k2;

        int positionSon;

        BNode\* parent = current->parent;

        for (int j = 0; j <= parent->count; j++) {

            if (parent->children[j] == current) {

                positionSon = j;

                break;

            }

        }

        if (positionSon == 0) {

            InsertToNode(\*parent->records[positionSon], current);

            LConnect(current, parent->children[positionSon + 1]);

            parent->children[positionSon + 1] = current;

            parent->children[positionSon] = nullptr;

            RemoveFromNode(criterion(\*parent->records[positionSon]), parent);

            if (current->count == 2 \* N) {

                while (current->count == 2 \* N) {

                    if (current == root) {

                        Restruct(current);

                        break;

                    }

                    else {

                        Restruct(current);

                        current = current->parent;

                    }

                }

            }

            else

                if (parent->count <= (N - 2)) Repair(parent);

        }

        else {

            if (positionSon == parent->count) {

                InsertToNode(\*parent->records[positionSon - 1], parent->children[positionSon - 1]);

                LConnect(parent->children[positionSon - 1], current);

                parent->children[positionSon] = parent->children[positionSon - 1];

                parent->children[positionSon - 1] = nullptr;

                RemoveFromNode(criterion(\*parent->records[positionSon - 1]), parent);

                BNode\* temp = parent->children[positionSon];

                if (current->count == 2 \* N) {

                    while (temp->count == 2 \* N) {

                        if (temp == root) {

                            Restruct(temp);

                            break;

                        }

                        else {

                            Restruct(temp);

                            temp = temp->parent;

                        }

                    }

                }

                else

                    if (parent->count <= (N - 2)) Repair(parent);

            }

            else {

                InsertToNode(\*parent->records[positionSon], current);

                LConnect(current, parent->children[positionSon + 1]);

                parent->children[positionSon + 1] = current;

                parent->children[positionSon] = nullptr;

                RemoveFromNode(criterion(\*parent->records[positionSon]), parent);

                if (current->count == 2 \* N) {

                    while (current->count == 2 \* N) {

                        if (current == root) {

                            Restruct(current);

                            break;

                        }

                        else {

                            Restruct(current);

                            current = current->parent;

                        }

                    }

                }

                else

                    if (parent->count <= (N - 2))

                        Repair(parent);

            }

        }

    }

    };

}

#define LIST Lists::List

#define CRITERION(indCriterion) criterions[indCriterion]

namespace Lists {

    template<int N>

    struct List {

    private:

        struct Node

        {

            Node\* nextNode;

            int index = 0;

            RECORD\* record = 0;

            Node(int ind, RECORD\* im) {

                nextNode = nullptr;

                index = ind;

                record = im;

            }

            Node() {

                nextNode = nullptr;

                index = 0;

                record = RECORD();

            }

            template <typename T>

            T GetValue(T(\*criterion)(RECORD&)) {

                return criterion(\*record);

            }

            bool operator==(const Node& node) {

                return record == node.record;

            }

            bool operator!=(const Node& node) {

                return !operator==(node);

            }

        };

    public:

        RECORD\* Find(unsigned int value, int criterionInd) {

            Node\* current = first[criterionInd + 1];

            while (current && current->GetValue(CRITERION(criterionInd)) != value) {

                current = current->nextNode;

            }

            if (current == nullptr)

                return new RECORD();

            return current->record;

        }

        List() {

            unsigned int (\*criterionsForList[])(RECORD&) = { MyTypes::criterionName, MyTypes::criterionHeight };

            static\_assert(N > 0 && N <= 6, "Количество критериев должно быть больше 0");

            for (size\_t i = 0; i < N; i++) {

                criterions[i] = criterionsForList[i];

                first[i] = nullptr;

                last[i] = nullptr;

            }

            first[N] = nullptr;

            last[N] = nullptr;

            count = 0;

        }

        int criterionInds[N];

        List(int criterions[N]) {

            static\_assert(N > 0 && N <= 6, "Количество критериев должно быть больше 0");

            for (size\_t i = 0; i < N; i++) {

                this->criterions[i] = MyTypes::criterions[criterions[i]];

                criterionInds[i] = criterions[i];

                first[i] = nullptr;

                last[i] = nullptr;

            }

            first[N] = nullptr;

            last[N] = nullptr;

            count = 0;

        }

        bool IsEmpty() {

            bool result = true;

            for (int i = 0; i < N+1; i++) {

                result = result && (first[i] == nullptr);

            }

            return result;

        }

        bool IsEmpty(int ind) {

            return first[ind] == nullptr;

        }

        void Push(RECORD\* value) {

            count++;

            Node\* newPointer = new Node(count, value);

            if (IsEmpty(0)) {

                first[0] = newPointer;

                last[0] = newPointer;

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

                    SortedPush(value, i+1);

                }

                return;

            }

            newPointer->nextNode = first[0];

            first[0] = newPointer;

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

                SortedPush(value, i+1);

            }

        }

        int count = 0;

        int Count() { return count; }

        void PushBack(RECORD\* value) {

            count++;

            Node\* newPointer = new Node(count, value);

            if (IsEmpty(0)) {

                first[0] = newPointer;

                last[0] = newPointer;

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

                    SortedPush(value, i + 1);

                }

                return;

            }

            last[0]->nextNode = newPointer;

            last[0] = newPointer;

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

                SortedPush(value, i+1);

            }

        }

        Node\* operator[] (const int index) {

            if (IsEmpty(0)) return nullptr;

            Node\* current = first[0];

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

                current = current->nextNode;

                if (!current) return nullptr;

            }

            return current;

        }

        void ReverseIter(int IndCriterion) {

            last[IndCriterion] = first[IndCriterion];

            Node\* buffer = GetReverseListIt(first[IndCriterion]);

            first[IndCriterion] = buffer;

        }

        void ReverseRec(int IndCriterion) {

            last[IndCriterion] = first[IndCriterion];

            Node\* buffer = GetReverseListRec(first[IndCriterion]);

            first[IndCriterion] = buffer;

        }

        void Remove(unsigned int value, MyTypes::Criterion criterion) {

            if (IsEmpty(0)) return;

            if (first[0] == last[0]) {

                for (size\_t i = 0; i < N + 1; i++)  {

                    delete first[i];

                    first[i] = nullptr;

                    last[i] = nullptr;

                }

                return;

            }

            if (first[0]->GetValue(criterion) == value) {

                RemoveFirst();

                return;

            }

            else if (last[0]->GetValue(criterion) == value) {

                RemoveLast();

                return;

            }

            Node\* slow = first[0];

            Node\* fast = first[0]->nextNode;

            while (fast && fast->GetValue(criterion) != value) {

                slow = fast;

                fast = fast->nextNode;

            }

            if (!fast) {

                std::cout << "This element does not exist" << std::endl;

                return;

            }

            RemoveNode(fast);

            slow->nextNode = fast->nextNode;

            delete fast;

        }

        void RemoveFirst() {

            if (IsEmpty(0)) return;

            Node\* node = first[0];

            first[0] = first[0]->nextNode;

            RemoveNode(node);

            delete node;

        }

        void RemoveLast() {

            if (IsEmpty(0)) return;

            Node\* current = first[0];

            while (current->nextNode != last[0]) {

                current = current->nextNode;

            }

            Node\* node = last[0];

            last[0] = current;

            RemoveNode(node);

            delete node;

        }

        void DeleteWithNumber(unsigned int value) {

            if (IsEmpty(0)) return;

            if (first[0] == last[0]) {

                for (size\_t i = 0; i < N + 1; i++) {

                    delete first[i];

                    first[i] = nullptr;

                    last[i] = nullptr;

                }

                return;

            }

            if (first[0]->record->number == value) {

                RemoveFirst();

                return;

            }

            else if (last[0]->record->number == value) {

                RemoveLast();

                return;

            }

            Node\* slow = first[0];

            Node\* fast = first[0]->nextNode;

            while (fast && fast->record->number != value) {

                slow = fast;

                fast = fast->nextNode;

            }

            if (!fast) {

                std::cout << "This element does not exist" << std::endl;

                return;

            }

            RemoveNode(fast);

            slow->nextNode = fast->nextNode;

            delete fast;

        }

        void DeleteData() {

            if (IsEmpty(0)) return;

            count = 0;

            for (int i = 0; i < N + 1;i++) {

                Node\* node = first[i];

                while (node != nullptr) {

                    Node\* deleted = node;

                    node = node->nextNode;

                    delete deleted;

                }

                last[i] = nullptr;

                first[i] = nullptr;

            }

        }

        void WriteList() {

            if (IsEmpty(0)) return;

            Node\* current = first[0];

            std::cout << "////////////////////////////////////////////////////////////////\n";

            std::cout << "Number/1 Name/2 Size/3 Width/4 Height/5 ColorDepth/6 Format///\n";

            while (current != nullptr)

            {

                if (current->record)

                    current->record->Write();

                std::cout << std::endl;

                current = current->nextNode;

                if (current) {

                    std::cout << "|" << std::endl;

                    std::cout << "V" << std::endl;

                }

            }

            std::cout << "////////////////////////////////////////////////////////////////\n";

        }

        void WriteList(int criterionInd, bool rise = true) {

            if (IsEmpty(criterionInd)) return;

            Node\* current;

            if (!rise)

                ReverseRec(rise);

            current = first[criterionInd];

            if (criterionInd == 0) {

                WriteList();

                return;

            }

            std::cout << "////////////////////////////////////////////////////////////////\n";

            std::cout << "Number/1 Name/2 Size/3 Width/4 Height/5 ColorDepth/6 Format///\n";

            while (current)

            {

                if(current->record != NULL)

                    current->record->Write();

                std::cout << std::endl;

                current = current->nextNode;

                if (current) {

                    for (int i = 0; i < criterionInds[criterionInd - 1] + 1; i++)

                        std::cout << "\t";

                    std::cout << "|" << std::endl;

                    for (int i = 0; i < criterionInds[criterionInd - 1] + 1; i++)

                        std::cout << "\t";

                    std::cout << "V" << std::endl;

                }

            }

            if (!rise)

                ReverseRec(criterionInd);

            std::cout << "////////////////////////////////////////////////////////////////\n";

        }

    private:

        void Push(RECORD\* value, int indCriterion) {

            Node\* newPointer = new Node(count, value);

            if (IsEmpty(indCriterion)) {

                first[indCriterion] = newPointer;

                last[indCriterion] = newPointer;

                return;

            }

            newPointer->nextNode = first[indCriterion];

            first[indCriterion] = newPointer;

        }

        void PushBack(RECORD\* value, int indCriterion) {

            Node\* newPointer = new Node(count, value);

            if (IsEmpty(indCriterion)) {

                first[indCriterion] = newPointer;

                last[indCriterion] = newPointer;

                return;

            }

            last[indCriterion]->nextNode = newPointer;

            last[indCriterion] = newPointer;

        }

        Node\* GetReverseListIt(Node\* root) {

            Node\* new\_root = 0;

            while (root) {

                Node\* next = root->nextNode;

                root->nextNode = new\_root;

                new\_root = root;

                root = next;

            }

            return new\_root;

        }

        Node\* GetReverseListRec(Node\* root) {

            if (root == nullptr || root->nextNode == nullptr)

                return root;

            Node\* rest = GetReverseListRec(root->nextNode);

            root->nextNode->nextNode = root;

            root->nextNode = nullptr;

            return rest;

        }

        void RemoveNode(Node\* node) {

            for (int i = 1; i < N + 1; i++) {

                if ((\*first[i]) == (\*node)) {

                    RemoveFirstNode(i);

                    continue;

                }

                else if ((\*last[i]) == (\*node)) {

                    RemoveLastNode(i);

                    continue;

                }

                Node\* slow = first[i];

                Node\* fast = first[i]->nextNode;

                while (fast && ((\*fast) != (\*node))) {

                    slow = fast;

                    fast = fast->nextNode;

                }

                slow->nextNode = fast->nextNode;

                delete fast;

            }

        }

        void RemoveFirstNode(int index) {

            if (IsEmpty(index)) return;

            Node\* node = first[index];

            first[index] = first[index]->nextNode;

            delete node;

        }

        void RemoveLastNode(int index) {

            if (IsEmpty(index)) return;

            Node\* current = first[index];

            while (current->nextNode != last[index]) {

                current = current->nextNode;

            }

            Node\* node = last[index];

            last[index] = current;

            delete node;

        }

        void SortedPush(RECORD\* value, int indCriterion) {

            unsigned int val = CRITERION(indCriterion - 1)(\*value);

            Node\* newPointer = new Node(count, value);

            if (IsEmpty(indCriterion)) {

                first[indCriterion] = newPointer;

                last[indCriterion] = newPointer;

                return;

            }

            if (val <= first[indCriterion]->GetValue(CRITERION(indCriterion - 1))) {

                Push(value, indCriterion);

                return;

            }

            if (val >= last[indCriterion]->GetValue(CRITERION(indCriterion - 1))) {

                PushBack(value, indCriterion);

                return;

            }

            Node\* slow = first[indCriterion];

            Node\* fast = first[indCriterion]->nextNode;

            while (fast && (val > fast->GetValue(CRITERION(indCriterion - 1)))) {

                slow = fast;

                fast = fast->nextNode;

            }

            slow->nextNode = newPointer;

            newPointer->nextNode = fast;

        }

        MyTypes::Criterion criterions[N];

        Node\* first[N + 1];

        Node\* last[N + 1];

    };

};

struct Table {

private:

    ARRAY array = ARRAY();

    LIST<2> list;

    TREE<2> tree;

    int\* indexesCrit = 0;

    MyTypes::Criterion criterions[2];

public:

    Table(int criterionInds[2], std::string filename) {

        array = ARRAY::GetArrayFromFile(filename);

        list = LIST<2>(criterionInds);

        tree = TREE<2>(criterionInds[0]);

        for (size\_t i = 0; i < 2; i++)

        {

            this->criterions[i] = MyTypes::criterions[criterionInds[i]];

        }

        for (int i = 0; i < array.Count(); i++) {

            list.PushBack(&array[i]);

            tree.Insert(array[i]);

        }

    }

    Table(int criterionInds[2],int length) {

        array = ARRAY::GetRandom(length);

        list = LIST<2>(criterionInds);

        tree = TREE<2>(criterionInds[0]);

        for (size\_t i = 0; i < 2; i++)

        {

            this->criterions[i] = MyTypes::criterions[criterionInds[i]];

        }

        for (int i = 0; i < array.Count(); i++) {

            list.PushBack(&array[i]);

            tree.Insert(array[i]);

        }

    }

    Table() {

    }

    Table(int criterionInds[2]) {

        array = ARRAY::GetArrayFromConsole();

        list = LIST<2>(criterionInds);

        tree = TREE<2>(criterionInds[0]);

        for (size\_t i = 0; i < 2; i++)

        {

            this->criterions[i] = MyTypes::criterions[criterionInds[i]];

        }

        for (int i = 0; i < array.Count(); i++) {

            list.PushBack(&array[i]);

            tree.Insert(array[i]);

        }

    }

    struct TablePTRRecord {

        void Write() {

            record.Write();

        }

        void SetName(std::string name);

        void SetSize(unsigned int size);

        void SetWidth(unsigned short width);

        void SetHeight(unsigned short height);

        void SetColorDepth(unsigned char colorDepth);

        void SetFormat(std::string format);

        TablePTRRecord(Table\* table, RECORD& record) :record(record), table(table) {};

    private:

        RECORD& record;

        Table\* table;

    };

    void SortAll();

    void WriteArray();

    void WriteSortArray(int criterion,bool rise = true);

    int FindInArray(unsigned int value, int criterion);

    TablePTRRecord operator [](int index);

    void DeleteRecordWhithArray(unsigned value, int criterion);

    void WriteSortArrayWithTree(bool rise = true);

    RECORD FindWithTree(unsigned int value);

    void DeleteRecordFromTree(unsigned int value);

    void WriteArrayWithList();

    void WriteSortArrayWithList(int criterion, bool rise = true);

    RECORD FindRecordWithList(unsigned int value, int criterion);

    void DeleteRecordFromList(unsigned int value, int criterion);

};

void Table::TablePTRRecord::SetName(std::string name) {

    record.name = name;

    table->SortAll();

}

void  Table::TablePTRRecord::SetSize(unsigned int size) {

    record.size = size;

    table->SortAll();

}

void  Table::TablePTRRecord::SetWidth(unsigned short width) {

    record.width = width;

    table->SortAll();

}

void  Table::TablePTRRecord::SetHeight(unsigned short height) {

    record.height = height;

    table->SortAll();

}

void  Table::TablePTRRecord::SetColorDepth(unsigned char colorDepth) {

    record.colorDepth = colorDepth;

    table->SortAll();

}

void Table::TablePTRRecord::SetFormat(std::string format) {

    record.format = format;

    table->SortAll();

}

void Table::SortAll()

{

    tree.DeleteData();

    list.DeleteData();

    for (int i = 0; i < array.Count(); i++) {

        list.PushBack(&array[i]);

        tree.Insert(array[i]);

    }

}

void Table::WriteArray()

{

    array.WriteArray();

}

void Table::WriteSortArray(int criterion, bool rise ) {

    array.SortIndexes(&indexesCrit, criterions[criterion], rise);

    for (size\_t i = 0; i < array.Count(); i++)

    {

        std::cout << std::to\_string(indexesCrit[i]) << " ";

    }

    std::cout << "\n";

    array.WriteArray(indexesCrit);

}

int  Table::FindInArray(unsigned int value, int criterion) {

    array.SortIndexes(&indexesCrit, criterions[criterion]);

    int result = array.FindBinary<unsigned int>(indexesCrit,value, criterions[criterion]);

    if (result == -1)

        std::cout << ("Такого записи с таким ключем нет - " + std::to\_string(value)) << "\n";

    return result;

}

Table::TablePTRRecord  Table::operator [](int index) {

    return TablePTRRecord(this,array[index]);

}

void  Table::DeleteRecordWhithArray(unsigned value, int criterion) {

    int index = FindInArray(value,criterion);

    if (index == -1) {

        std::cout << "Элемент удалить невозможно\n";

        return;

    }

    RECORD elem = array[index];

    array.DeleteWithNumber(elem.number);

    array.RecoverIndexes();

    SortAll();

}

void  Table::WriteSortArrayWithTree(bool rise) {

    tree.WriteTree(rise);

}

RECORD Table::FindWithTree(unsigned int value) {

    RECORD elem = tree.Find(value);

    if (elem == RECORD()) {

        std::cout << "Такого ключа в дереве нет\n";

    }

    return elem;

}

void  Table::DeleteRecordFromTree(unsigned int value) {

    tree.Delete(value);

    array.RecoverIndexes();

}

void  Table::WriteArrayWithList() {

    list.WriteList();

}

void  Table::WriteSortArrayWithList(int criterion, bool rise) {

    list.WriteList(criterion,rise);

}

RECORD Table::FindRecordWithList(unsigned int value, int criterion) {

    RECORD elem = \*list.Find(value, criterion);

    if (elem == RECORD()) {

        std::cout << "Такого ключа в списке нет\n";

    }

    return elem;

}

void Table::DeleteRecordFromList(unsigned int value, int criterion) {

    list.Remove(value,criterions[criterion]);

    array.RecoverIndexes();

}

#include <fstream>

#define CONSOLE 1

#define FILE  2

namespace Arrays {

    MyArray::MyArray()

    {

        length = 0;

        records = 0;

    }

    MyArray::MyArray(size\_t len)

    {

        length = len;

        records = new RECORD[len];

    }

    MyArray::MyArray(const MyArray& array)

    {

        length = array.length;

        records = array.records;

    }

    MyArray MyArray::GetRandom(size\_t length)

    {

        MyArray result(length);

        for (size\_t i = 0; i < length; i++)

        {

            result[i] = DataManage::InputManager::GetRandomRecord();

            result[i].number = i + 1;

        }

        return result;

    }

    MyArray MyArray::GetArrayFromFile(std::string fileName) {

        std::ifstream file(fileName);

        int len = 0;

        std::string s;

        if (file.is\_open()) {

            while (std::getline(file, s)) {

                (len)++;

            }

        }

        else

            return MyArray();

        file.close();

        file = std::ifstream(fileName);

        MyArray result(len);

        if (len == 0)

            throw "Не удалось открыть файл";

        int i = 0;

        while (i < len) {

            std::getline(file, s);

            result[i] = DataManage::InputManager::GetRecord(FILE,s,'\t');

            i++;

        }

        file.close();

        return result;

    }

    MyArray MyArray::GetArrayFromConsole()

    {

        std::cout << "Введите длину массива" << "\n";

        std::string line = "";

        std::getline(std::cin, line);

        size\_t length = std::stoull(line);

        MyArray result(length);

        for (size\_t i = 0; i < length; i++)

        {

            result[i] = DataManage::InputManager::GetRecord(CONSOLE);

            result[i].number = i + 1;

        }

        return result;

    }

    void MyArray::WriteArray() {

        std::cout << "////////////////////////////////////////////////////////////////\n";

        std::cout << "Number/1 Name/2 Size/3 Width/4 Height/5 ColorDepth/6 Format///\n";

        for (size\_t i = 0; i < length; i++) {

            records[i].Write();

            std::cout << "\n";

        }

        std::cout << "////////////////////////////////////////////////////////////////\n";

    }

    void MyArray::WriteArray(int\* indexes) {

        std::cout << "////////////////////////////////////////////////////////////////\n";

        std::cout << "Number/1 Name/2 Size/3 Width/4 Height/5 ColorDepth/6 Format///\n";

        for (size\_t i = 0; i < length; i++) {

            records[indexes[i]].Write();

            std::cout << "\n";

        }

        std::cout << "////////////////////////////////////////////////////////////////\n";

    }

    void MyArray::Reconstruct() {

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

            RECORD buff = records[records[i].number - 1];

            records[records[i].number - 1] = records[i];

            records[i] = buff;

        }

    }

    void MyArray::Sort(size\_t indStart, size\_t indEnd, MyTypes::Criterion criterion, bool rise, bool quick) {

        if (!quick)

            for (size\_t i = indStart; i < indEnd - 1; i++) {

                for (size\_t j = i + 1; j < indEnd; j++) {

                    if (criterion(records[i]) > criterion(records[j]) == rise) {

                        auto buff = records[i];

                        records[i] = records[j];

                        records[j] = buff;

                    }

                }

            }

        else {

            QuickSort(records + indStart, indEnd - indStart , criterion, rise);

        }

    }

    void MyArray::Sort(MyTypes::Criterion criterion, bool rise,bool quick ) {

        if(!quick)

        for (int i = 0; i < length - 1; i++) {

            for (int j = i + 1; j < length; j++) {

                if (criterion(records[i]) > criterion(records[j]) == rise) {

                    auto buff = records[i];

                    records[i] = records[j];

                    records[j] = buff;

                }

            }

        }

        else {

            QuickSort(records,length,criterion,rise);

        }

    }

    void MyArray::DeleteWithNumber(size\_t value) {

        size\_t ind = value - 1;

        InBorder(ind);

        RECORD\* result = new RECORD[length - 1];

        size\_t j = 0;

        for (size\_t i = 0; i < ind; i++)

        {

            result[j++] = records[i];

        }

        for (size\_t i = ind + 1; i < length; i++)

        {

            result[j++] = records[i];

        }

        delete[] records;

        records = result;

        length -= 1;

        RecoverIndexes();

    }

    void MyArray::QuickSort(RECORD\* array, size\_t len, MyTypes::Criterion criterion,bool rise) {

        long long i = 0;

        long long j = len - 1;

        unsigned int mid = criterion(array[len / 2]);

        do {

            if (rise) {

                while (criterion(array[i]) < mid) {

                    i++;

                }

                while (criterion(array[j]) > mid) {

                    j--;

                }

                if (i <= j) {

                    RECORD tmp = array[i];

                    array[i] = array[j];

                    array[j] = tmp;

                    i++;

                    j--;

                }

            }

            else {

                while (criterion(array[i]) > mid) {

                    i++;

                }

                while (criterion(array[j]) < mid) {

                    j--;

                }

                if (i <= j) {

                    RECORD tmp = array[i];

                    array[i] = array[j];

                    array[j] = tmp;

                    i++;

                    j--;

                }

            }

        } while (i <= j);

        if (j > 0) {

            QuickSort(array,j + 1,criterion,rise);

        }

        if (i < len) {

            QuickSort(&array[i], len - i, criterion,rise);

        }

    }

    void MyArray::GetSortIndexes(int\* indexes, MyTypes::Criterion criterion, bool rise) {

        for (int i = 0; i < length - 1; i++) {

            for (int j = i + 1; j < length; j++) {

                if ((criterion(records[indexes[i]]) > criterion(records[indexes[j]])) == rise) {

                    auto buff = indexes[i];

                    indexes[i] = indexes[j];

                    indexes[j] = buff;

                }

            }

        }

    }

    void MyArray::SortIndexes(int\*\* indexes, MyTypes::Criterion criterion, bool rise, bool quick) {

        delete[](\*indexes);

        (\*indexes) = new int[Count()];

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

            (\*indexes)[i] = i;

        }

        if (quick) {

            QuickGetSortIndexes((\*indexes), Count(), criterion, rise);

        }

        else {

            GetSortIndexes((\*indexes), criterion, rise);

        }

    }

    void MyArray::QuickGetSortIndexes(int\* indexes, size\_t len, MyTypes::Criterion criterion, bool rise) {

        long long i = 0;

        long long j = len - 1;

        unsigned int mid = criterion(records[indexes[len / 2]]);

        do {

            if (rise) {

                while (criterion(records[indexes[i]]) < mid) {

                    i++;

                }

                while (criterion(records[indexes[j]]) > mid) {

                    j--;

                }

                if (i <= j) {

                    int tmp = indexes[i];

                    indexes[i] = indexes[j];

                    indexes[j] = tmp;

                    i++;

                    j--;

                }

            }

            else {

                while (criterion(records[indexes[i]]) > mid) {

                    i++;

                }

                while (criterion(records[indexes[j]]) < mid) {

                    j--;

                }

                if (i <= j) {

                    int tmp = indexes[i];

                    indexes[i] = indexes[j];

                    indexes[j] = tmp;

                    i++;

                    j--;

                }

            }

        } while (i <= j);

        if (j > 0) {

            QuickGetSortIndexes(indexes, j + 1, criterion, rise);

        }

        if (i < len) {

            QuickGetSortIndexes(&indexes[i], len - i, criterion, rise);

        }

    }

    int MyArray::Find(RECORD value) {

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

            if (records[i] == value)

                return i;

        return -1;

    }

    void MyArray::RecoverIndexes(bool rise) {

        int\* indexes = 0;

        SortIndexes(&indexes, [](RECORD& image) {return (unsigned int)image.number; }, rise);

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

            records[indexes[i]].number = i + 1;

        }

        delete[] indexes;

    }

    void MyArray::DeleteFirstElem(RECORD value) {

        int ind = Find(value);

        if (ind == -1)

            return;

        RECORD\* currentArray = new RECORD[length - 1];

        int j = 0;

        for (int i = 0; i < ind; i++)

        {

            currentArray[j++] = records[i];

        }

        for (int i = ind + 1; i < length; i++)

        {

            currentArray[j++] = records[i];

        }

        delete[] records;

        records = currentArray;

        length -= 1;

        RecoverIndexes();

    }

    void MyArray::AddElem(RECORD value) {

        size\_t newLen = 1 + length;

        ReSize(newLen);

        records[length] = value;

        length = newLen;

    }

    void MyArray::AddElems(int count) {

        size\_t newLen = count + length;

        RECORD\* result = new RECORD[newLen];

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

            result[i] = records[i];

        }

        for (int i = length; i < newLen; i++) {

            result[i] = DataManage::InputManager::GetRecord(CONSOLE);

        }

        delete[] records;

        records = result;

        length = newLen;

    }

    void MyArray::AddElems(std::string fileName) {

        MyArray addRecords = GetArrayFromFile(fileName);

        size\_t newLen = addRecords.Count() + length;

        RECORD\* result = new RECORD[newLen];

        int count = 1;

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

            result[i] = records[i];

            count++;

        }

        for (int i = 0; i < addRecords.Count(); i++) {

            result[i + length] = addRecords[i];

            result[i + length].number = count++;

        }

        delete[] records;

        records = result;

        length = newLen;

    }

    void MyArray::ReSize(size\_t newLen) {

        RECORD\* newRecords = new RECORD[newLen];

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

            newRecords[i] = records[i];

        }

        delete[] records;

        records = newRecords;

    }

}

#include <Windows.h>

int ReadValueUInt() {

    int a = 0;

    do {

        std::cin >> a;

        std::cin.ignore();

        if (a < 0)

            std::cout << "Число должно быть больше либо равно нулю\n";

    } while (a < 0);

    return a;

}

std::string ReadString() {

    std::string result = "";

    std::getline(std::cin,result);

    return result;

}

void print(std::string str = "", char end = '\n') {

    std::cout << str << end;

}

int main()

{

    SetConsoleCP(1251);

    SetConsoleOutputCP(1251);

    srand(time(NULL));

    int criterionInds[2] = { 3,0 };

    setlocale(LC\_ALL, "Rus");

    int action = 1;

    bool flag = false;

    print("Введите 1 для заполения из файла");

    print("Введите 2 для заполнения из консоли");

    print("Введите 3 для заполнения случайными полями");

    Table table = Table(criterionInds, "Data.txt");

    do {

        action = ReadValueUInt();

        switch (action)

        {

        case 1:

            flag = true;

            break;

        case 2:

            flag = true;

            table = Table(criterionInds);

            table.SortAll();

            break;

        case 3:

            flag = true;

            print("Введите длину массива");

            table = Table(criterionInds, ReadValueUInt());

            table.SortAll();

            break;

        default:

            print("Введите цифру нормально");

            break;

        }

    } while (!flag);

    flag = true;

    ARRAY array = ARRAY::GetRandom(10);

    TABLE hashTable = TABLE(array.Count());

    for (size\_t i = 0; i < array.Count(); i++)

    {

        hashTable.AddPair(array[i].name,array[i]);

    }

    hashTable.GetValue(array[3].name).Write();

    print();

    hashTable.GetValue("sasdasd").Write();

    print();

    hashTable.GetValue("Бабочка").Write();

    print();

    hashTable.AddPair("asd", RECORD(2,"asd", 2, 2, 2,2, "dsds"));

    hashTable.GetValue("asd").Write();

    print("Исходная таблица");

    table.WriteArray();

    print("Лист");

    table.WriteArrayWithList();

    print("Дерево");

    table.WriteSortArrayWithTree();

    system("pause");

    RECORD record = RECORD();

    int index = 0;

    int value = 0;

    int findIndex = 0;

    std::string line;

    system("pause");

    while (action != -1) {

        system("cls");

        print("Первый критерий - высота, он используется в дереве");

        print("Второй критерий - Имя");

        print("1. работа с массивом");

        print("2. работа с деревом");

        print("3. работа со списком");

        do {

            if(flag)

                action = ReadValueUInt();

            switch (action)

            {

            case 1:

                system("cls");

                table.WriteArray();

                print("1. - Отсортировать по имени");

                print("2. - Отсортировать по высоте");

                print("3. - удалить элемент");

                print("4. - найти элемент");

                do {

                    action = ReadValueUInt();

                    switch (action)

                    {

                    case 1:

                        table.WriteSortArray(1);

                        break;

                    case 2:

                        table.WriteSortArray(0);

                        break;

                    case 3:

                        print("Введите индекс ключа (0, 1) и значение которое хотите удалить");

                        index = ReadValueUInt();

                        value = ReadValueUInt();

                        if (index > 1)

                            break;

                        table.DeleteRecordWhithArray(value,index);

                        break;

                    case 4:

                        print("Введите индекс ключа (0, 1) и значение которое хотите найти");

                        index = ReadValueUInt();

                        if (index > 1)

                            break;

                        if (index == 0) {

                            value = ReadValueUInt();

                            findIndex = table.FindInArray(value, index);

                        }

                        if (index == 1) {

                            line = ReadString();

                            findIndex = table.FindInArray(MyTypes::GetIntFromString(line), index);

                        }

                        if (findIndex != -1) {

                            table[findIndex].Write();

                            print();

                        }

                        else{

                            print("Такого элемента нет");

                        }

                        break;

                    default:

                        flag = true;

                        break;

                    }

                } while (!flag);

                system("pause");

                break;

            case 2:

                system("cls");

                table.WriteSortArrayWithTree();

                print("1. - удалить элемент");

                print("2. - найти элемент");

                do {

                    action = ReadValueUInt();

                    switch (action)

                    {

                    case 1:

                        print("Введите значение которое хотите удалить (Высота)");

                        value = ReadValueUInt();

                        table.DeleteRecordFromTree(value);

                        break;

                    case 2:

                        print("Введите значение которое хотите найти(Высота)");

                        value = ReadValueUInt();

                        record = table.FindWithTree(value);

                        record.Write();

                        if (record == RECORD())

                            print("Такой записи нет в дереве");

                        print();

                        break;

                    default:

                        flag = true;

                        break;

                    }

                } while (!flag);

                system("pause");

                break;

            case 3:

                system("cls");

                table.WriteArrayWithList();

                print("1. - Отсортировать по имени");

                print("2. - Отсортировать по высоте");

                print("3. - удалить элемент");

                print("4. - найти элемент");

                do {

                    action = ReadValueUInt();

                    switch (action)

                    {

                    case 1:

                        table.WriteSortArrayWithList(2);

                        break;

                    case 2:

                        table.WriteSortArrayWithList(1);

                        break;

                    case 3:

                        print("Введите индекс ключа (0, 1) и значение которое хотите удалить");

                        index = ReadValueUInt();

                        value = ReadValueUInt();

                        if (index > 1)

                            break;

                        table.DeleteRecordFromList(value, index);

                        break;

                    case 4:

                        print("Введите индекс ключа (0, 1) и значение которое хотите найти");

                        index = ReadValueUInt();

                        if (index > 1)

                            break;

                        if (index == 0) {

                            value = ReadValueUInt();

                            record = table.FindRecordWithList(value, index);

                        }

                        if (index == 1) {

                            line = ReadString();

                            record = table.FindRecordWithList(MyTypes::GetIntFromString(line), index);

                        }

                        record.Write();

                        if (record == RECORD())

                            print("Такой записи нет в листе");

                        break;

                    default:

                        flag = true;

                        break;

                    }

                } while (!flag);

                system("pause");

                break;

            default:

                flag = true;

                break;

            }

        } while (!flag);

    }

}