МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

Національний аерокосмічний університет ім. М.Є. Жуковського

«Харківський авіаційний інститут»

Факультет програмної інженерії та бізнесу

Кафедра інженерії програмного забезпечення

**Курсова робота**

з дисципліни «Алгоритми та структури даних»

(назва дисципліни)

за темою: «Красно червоні дерева»

Виконав: студент \_\_2\_\_ курсу групи \_\_\_621п\_\_\_\_\_\_\_

напрямок підготовки (спеціальності)

6.121 « Інженерія програмного забезпечення» .

(шифр і назва напрямку підготовки (спеціальності))

Слюсар Антон Андрійович .

(прізвище та ініціали)

Прийняв: ст. викл. Мокляк Н.Г.

(посада, науковий ступінь, прізвище та ініціали)

Національна шкала: \_\_\_\_\_\_\_\_\_\_

Кількість балів: \_\_\_\_\_\_\_\_\_\_\_

Оцінка: ECTS \_\_\_\_\_\_\_\_\_\_

Члени комісії \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ (підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(підпис) (прізвище та ініціали

Харків – 2018 рік

Оглавление

[ВВЕДЕНИЕ 4](#_Toc512171794)

[1 ПОСТАНОВКА ЗАДАЧИ 5](#_Toc512171795)

[2 ТЕОРЕТИЧЕСКАЯ ИНФОРМАЦИЯ 6](#_Toc512171796)

[2.1 Деревья. Красно - черные деревья 6](#_Toc512171797)

[2.2 Свойства красно - черных деревьев 7](#_Toc512171798)

[2.3 Операции красно - черного дерева 8](#_Toc512171799)

[2.4 Вставка элемента в красно - черное дерево 9](#_Toc512171800)

[2.5 Удаление элемента из красно - черного дерева 10](#_Toc512171801)

[2.6 Поиск в красно - черных деревьях 12](#_Toc512171802)

[3 СТРУКТУРНАЯ СХЕМА ЗАДАЧИ 13](#_Toc512171803)

[4 МОДЕЛИРОВАНИЕ ПО ВИЗУАЛИЗАЦИИ МЕТОДОВ ВСТАВКИ, ПОИСКА И УДАЛЕНИЯ В КРАСНО – ЧЕРНЫХ ДЕРЕВЬЯХ 14](#_Toc512171804)

[5 ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ 16](#_Toc512171805)

[5.1 Входные данные 16](#_Toc512171806)

[5.1.1 Входные данные 16](#_Toc512171807)

[5.1.2 Входные формы 17](#_Toc512171808)

[5.2 ВЫХОДНЫЕ ДАННЫЕ 18](#_Toc512171809)

[5.2.1 Выходные данные 18](#_Toc512171810)

[5.2.2 Выходные формы 18](#_Toc512171811)

[6 АНОМАЛИИ 21](#_Toc512171812)

[7 ФУНКЦИОНАЛЬНЫЕ ТЕСТЫ 22](#_Toc512171813)

[8 АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ ПО ПО ВИЗУАЛИЗАЦИИ МЕТОДОВ ВСТАВКИ, ПОИСКА И УДАЛЕНИЯ В КРАСНО – ЧЕРНЫХ ДЕРЕВЬЯХ 24](#_Toc512171814)

[8.1 Общая структура классов 24](#_Toc512171815)

[8.2 Подробное описание классов 26](#_Toc512171816)

[8.2.1 Класс RBTreeNode: 26](#_Toc512171817)

[Таблица 7.2 – Поля и методы класса RBTreeNode 27](#_Toc512171818)

[8.2.2 Класс RBTree 28](#_Toc512171819)

[Таблица 7.2 – Поля и методы класса RBTree 28](#_Toc512171820)

[9 НИСХОДЯЩЕЕ ПРОЕКТИРОВАНИЕ 30](#_Toc512171821)

[9.1 Метод InsertNode – метод для добавления узла в дерево: 30](#_Toc512171822)

[9.2 Метод DeleteNode – метод удаления узла из красно–черного дерева. 32](#_Toc512171823)

[9.3 Метод Find – метод для поиска узла в красно–черном дерева. 35](#_Toc512171824)

[9.4 Метод RotateRight – метод для вращения поддерева вправо. 36](#_Toc512171825)

[10 ЛОГИЧЕСКАЯ МОДЕЛЬ ПРОГРАММЫ 40](#_Toc512171826)

[ВЫВОД 42](#_Toc512171827)

[ПЕРЕЧЕНЬ ССЫЛОК 44](#_Toc512171828)

[*ПРИЛОЖЕНИЕ А* 45](#_Toc512171829)

[*ПРИЛОЖЕНИЕ Б* 60](#_Toc512171830)

[*ПРИЛОЖЕНИЕ В* 61](#_Toc512171831)

# ВВЕДЕНИЕ

В данном курсовом проекте рассматривается разновидность бинарных поисковых деревьев – красно - черные деревья, которые также относятся к сбалансированным деревьям.

Красно - чёрное дерево (англ. Red–Black–Tree) – это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева – «цвета». Этот атрибут может принимать одно из двух возможных значений – «чёрный» или «красный».

Красно - черные деревья используются для решения самых разных задач, например, контейнер map и set в реализации библиотеки STL языка C++, класс java.util.TreeMap и java.util.TreeSet в языке Java, класс SortedDictionary и SortedSet, в языке C#, а также в ядре Linux (для организации очередей запросов, в файловой системе ext3).

Популярность красно–чёрных деревьев связана с тем, что на них часто достигается подходящий баланс между степенью сбалансированности и сложностью поддержки сбалансированности. В частности, при сравнении с идеально сбалансированными деревьями часто обнаруживается, что последние имеют слишком жесткое условие сбалансированности и при выполнении операций удаления из дерева много времени тратится на поддержание необходимой сбалансированности [1].

# ПОСТАНОВКА ЗАДАЧИ

**Цель работы**

Разработать программу с визуальной демонстрацией этапов вставки и удаления элементов в красно–черном дереве.

Для достижения цели необходимо решить следующие задачи:

**Задачи:**

1. Выполнить обзор и анализ существующих операций над красно–черными деревьями (добавление, удаление, поиск).
2. Разработка физической модели представления красно–черного дерева.
3. Согласно своему варианту создать класс с полями и методами, для реализации необходимых алгоритмов. Разрабатываемую структуру данных реализовать в виде класса.

# ТЕОРЕТИЧЕСКАЯ ИНФОРМАЦИЯ

## Деревья. Красно - черные деревья

Дерево ­– одна из наиболее широко распространённых структур данных в информатике, эмулирующая древовидную структуру в виде набора связанных узлов. Является связным графом, не содержащим циклы. Большинство источников также добавляют условие на то, что рёбра графа не должны быть ориентированными. В дополнение к этим трём ограничениям, в некоторых источниках указывается, что рёбра графа не должны быть взвешенными [2].

* Корневой узел – самый верхний узел дерева   
  (узел 8 на рисунке 2 .1).
* Корень – одна из вершин, по желанию наблюдателя.
* Лист, листовой или терминальный узел – узел, не имеющий дочерних элементов (узлы 1, 4, 7, 13).
* Внутренний узел – любой узел дерева, имеющий потомков, и таким образом, не являющийся листовым узлом (3, 6, 10, 14 на рисунке 2.1).
* Дерево считается ориентированным, если в корень не заходит ни одно ребро.
* Полный сцепленный ключ – идентификатор записи, который образуется путём конкатенации всех ключей экземпляров родительских записей (групп).

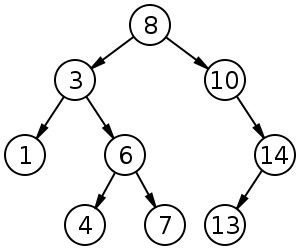


Рисунок 2.1 – Пример простого дерева

Красно–чёрное дерево (англ. Red–black tree, RB–Tree) ­– это одно из самобалансирующихся двоичных деревьев поиска, гарантирующих логарифмический рост высоты дерева от числа узлов и быстро выполняющее основные операции дерева поиска: добавление, удаление и поиск узла. Сбалансированность достигается за счёт введения дополнительного атрибута узла дерева – «цвета». Этот атрибут может принимать одно из двух возможных значений – «чёрный» или «красный».

Изобретателем красно - чёрного дерева считают немца Рудольфа Байера. Название «красно - чёрное дерево» структура данных получила в статье Л. Гимпаса и Р. Седжвика (1978). В журнале были доступны две краски (красная и чёрная) и дополнительный бит, «прикреплявшийся» к каждому из узлов, обозначался цветом.

## Свойства красно - черных деревьев

Красно - чёрное дерево ­­– двоичное дерево поиска, в котором каждый узел имеет атрибут цвет, принимающий значения красный или черный. В дополнение к обычным требованиям, налагаемым на двоичные деревья поиска, к красно–чёрным деревьям применяются следующие требования:

1. Каждый узел является красным или черным.

2. Корень дерева является черным.

3. Каждый лист дерева (NIL) является черным.

4. Если узел – красный, то оба его дочерних узла – черные.

5. Всякий путь от корня дерева к внешней вершине (листу) содержит одно и то же число черных вершин (постоянство чёрной высоты дерева по всем ветвям).

Эти ограничения реализуют критическое свойство красно–черных деревьев: путь от корня до самого дальнего листа не более чем в два раза длиннее пути от корня до ближайшего листа (если дальний лист расположен на 3–м уровне). Результатом является то, что дерево примерно сбалансировано. Так как такие операции как вставка, удаление и поиск значений требуют в худшем случае времени, пропорционального длине дерева, эта теоретическая верхняя граница высоты позволяет красно–чёрным деревьям быть более эффективными в худшем случае, чем обычные двоичные деревья поиска.

Чтобы понять, почему это гарантируется, достаточно рассмотреть эффект свойств 4 и 5 вместе. Пусть для красно - чёрного дерева T число черных узлов в свойстве 5 равно B. Тогда кратчайший возможный путь от корня дерева T до любого листового узла содержит B черных узлов. Более длинный возможный путь может быть построен путем включения красных узлов. Однако, свойство 4 не позволяет вставить несколько красных узлов подряд. Поэтому самый длинный возможный путь состоит из 2B узлов, попеременно красных и черных. Любой максимальный путь имеет одинаковое число черных узлов (по свойству 5), следовательно, не существует пути, более чем вдвое длинного, чем любой другой путь.

Во многих реализациях структуры дерева возможно, чтобы узел имел только одного потомка и листовой узел содержал данные. В этих предположениях реализовать красно - чёрное дерево возможно, но изменятся несколько свойств и алгоритм усложнится. По этой причине данная статья использует «фиктивные листовые узлы», которые не содержат данных и просто служат для указания, где дерево заканчивается. Эти узлы часто опускаются при графическом изображении, в результате дерево выглядит противоречиво с вышеизложенными принципами, но на самом деле противоречия нет. Следствием этого является то, что все внутренние (не являющиеся листовыми) узлы имеют два потомка, хотя один из них может быть нулевым листом. Свойство 5 гарантирует, что красный узел обязан иметь в качестве потомков либо два черных нулевых листа, либо два черных внутренних узла. Для чёрного узла с одним потомком ну левым листовым узлом и другим потомком, не являющимся таковым, свойства 3, 4 и 5 гарантируют, что последний должен быть красным узлом с двумя черными нулевыми листьями в качестве потомков (см. рисунок 2.2).

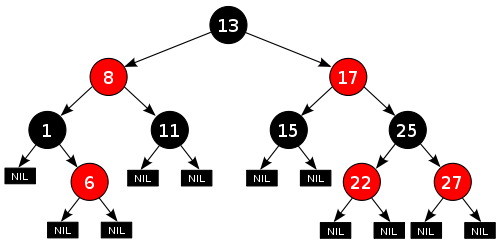


Рисунок 2.2 – Пример красно - черного дерева

## Операции красно - черного дерева

**Операции чтения для красно - чёрного дерева ничем не отличаются от иных для бинарного дерева поиска, потому что любое красно - чёрное дерево является особым случаем обычного бинарного дерева поиска. Однако непосредственный результат вставки или удаления может привести к нарушению свойств красно–черных деревьев. Восстановление свойств требует небольшого (O(log n) или O(1)) числа операций смены цветов (которая на практике очень быстрая) и не более чем трех поворотов дерева (для вставки – не более двух). Хотя вставка и удаление сложны, их трудоемкость остается O(log n).**

## Вставка элемента в красно - черное дерево

Каждый элемент вставляется вместо листа, поэтому для выбора места вставки идём от корня до тех пор, пока указатель на следующего сына не станет nil (то есть этот сын – лист). Вставляем вместо него новый элемент с nil - потомками и красным цветом. Теперь проверяем балансировку. Если отец нового элемента красный, то достаточно рассмотреть только два случая:

Ситуация 1: "Дядя" этого узла тоже красный. Тогда просто перекрашиваем "отца" и "дядю" в чёрный цвет, а "деда" – в красный. Проверяем, не нарушает ли он теперь балансировку. Если в результате этих перекрашиваний мы дойдём до корня, то в нём в любом случае ставим чёрный цвет (см. рисунок 2.3).

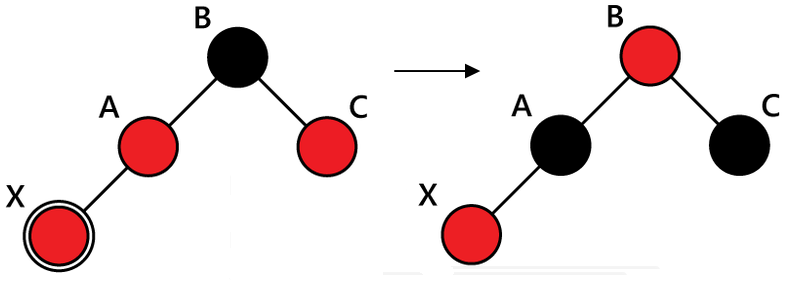


Рисунок 2.3 – Ситуация 1: «Перекраска узлов»

Ситуация 2: "Дядя" чёрный. Если выполнить только перекрашивание, то может нарушиться постоянство чёрной высоты дерева по всем ветвям. Поэтому выполняем поворот. Если добавляемый узел был правым потомком, то необходимо сначала выполнить левое вращение, которое сделает его левым потомком (см. рисунок 2.4).

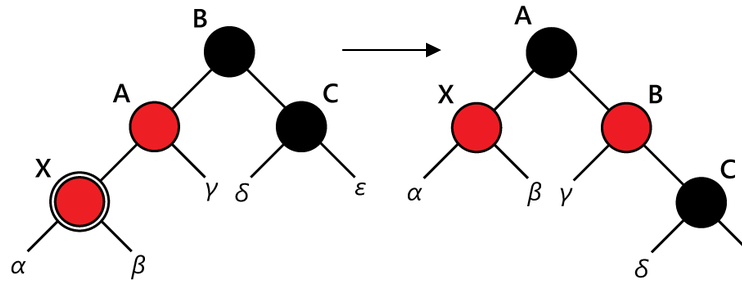


Рисунок 2.4 – Ситуация 2: «Использование поворотов»

## Удаление элемента из красно - черного дерева

При удалении вершины могут возникнуть три случая в зависимости от количества её детей:

1. Если у вершины нет детей, то изменяем указатель на неё у родителя на nil.
2. Если у неё только один ребёнок, то делаем у родителя ссылку на него вместо этой вершины.
3. Если же имеются оба ребёнка, то находим вершину со следующим значением ключа. У такой вершины нет левого ребёнка. Удаляем уже эту вершину описанным во втором пункте способом, скопировав её ключ в изначальную вершину.

Проверим балансировку дерева. Так как при удалении красной вершины свойства дерева не нарушаются, то восстановление балансировки потребуется только при удалении чёрной. Рассмотрим ребёнка удалённой вершины.

Ситуация 1. Если брат этого ребёнка красный, то делаем вращение вокруг ребра между отцом и братом, тогда брат становится родителем отца. Красим его в чёрный, а отца – в красный цвет (см. рисунок 2.5).

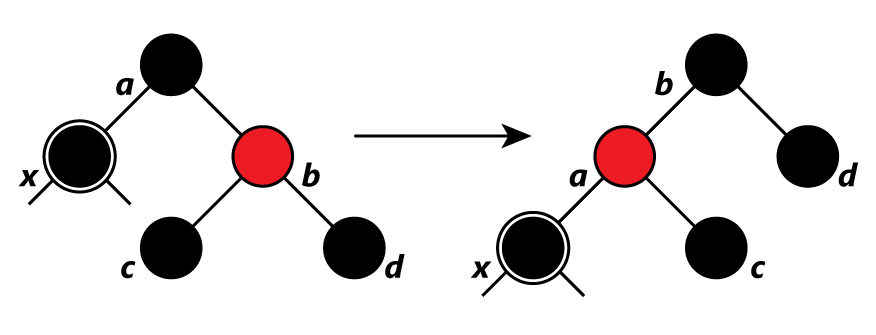


Рисунок 2.5 – Ситуация :1 «Вращение вокруг ребра между отцом и братом»

Ситуация 2. Если брат текущей вершины был чёрным, то получаем три случая:

* Оба ребёнка у брата чёрные. Красим брата в красный цвет и рассматриваем далее отца вершины (см. рисунок 2.6).

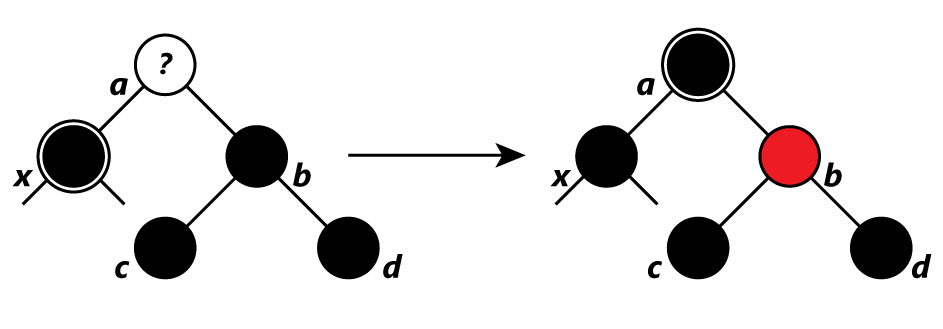


Рисунок 2.6 – Ситуация 2: «Первый случай»

* Если у брата правый ребёнок чёрный, а левый красный, то перекрашиваем брата и его левого сына и делаем вращение (см. рисунок 2.7).

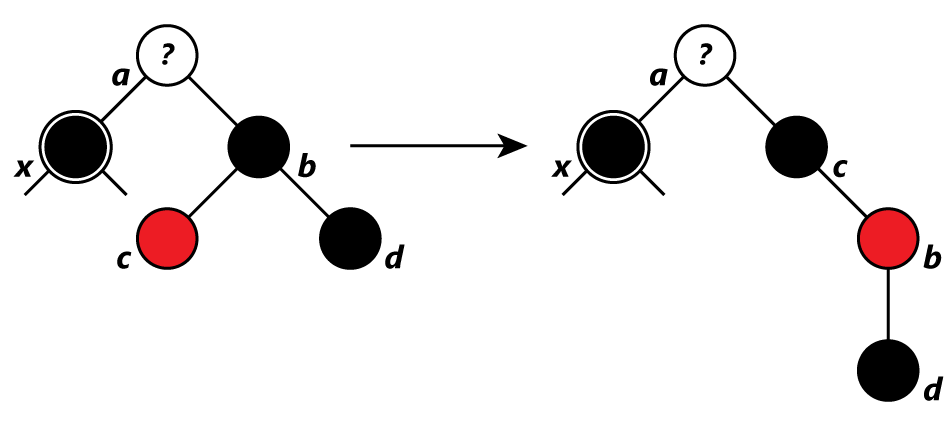


Рисунок 2.7 – Ситуация 2: «Второй случай»

* Если у брата правый ребёнок красный, то перекрашиваем брата в цвет отца, его ребёнка и отца – в чёрный, делаем вращение и выходим из алгоритма. Продолжаем тот же алгоритм, пока текущая вершина чёрная и мы не дошли до корня дерева (см. рисунок 2.8).

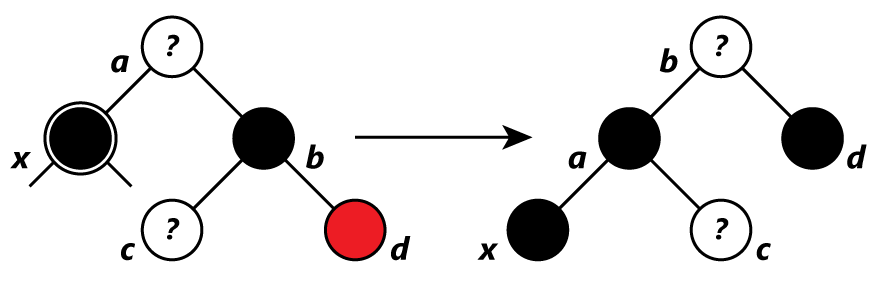


Рисунок 2.8 – Ситуация 2 «Третий случай»

Из рассмотренных случаев ясно, что при удалении выполняется не более трёх вращений.

## Поиск в красно - черных деревьях

Операция поиска получает на вход заданный ключ k и указатель t на корень поддерева, в котором производится поиск. Операция возвращает данные с ключом k или сообщение об ошибке, если узла с таким ключом нет в дереве.

В процессе поиска операция движется от корня, сравнивая ключ k с ключом key, хранящимся в текущем узле. Если они равны, или адрес узла равен nil, то поиск заканчивается.

Если k < key[t], то поиск продолжается в левом поддереве узла t (ключ может быть только там в силу свойства упорядоченности). Если k>key[t], то поиск продолжается в правом поддереве.

# СТРУКТУРНАЯ СХЕМА ЗАДАЧИ

Для решения задачи визуализации этапов вставки и удаления элементов в красно-черном дереве необходимо разбить на подзадачи. Структурная схема разбиения задачи «Визуализации этапов вставки и удаления элементов в красно-черном дереве» на более мелкие задачи представлена на рисунке 3.1.



Рисунок 3.1 – Структурная схема задачи

# МОДЕЛИРОВАНИЕ ПО ВИЗУАЛИЗАЦИИ МЕТОДОВ ВСТАВКИ, ПОИСКА И УДАЛЕНИЯ В КРАСНО – ЧЕРНЫХ ДЕРЕВЬЯХ

Для реализации поставленной задачи необходимо создать два класса (см. рисунок 3.1):

1. RBTreeNode – класс, который реализует структуру узла   
   красно - черного дерева (см. рисунок 3.2).
2. RBTree – класс, который реализует структуру красно - черного дерева (см. рисунок 3.3).

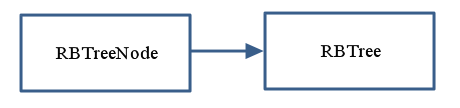


Рисунок 3.1 ­– Классы моделируемого ПО

Класс **RBTreeNode** должен представлять собой структуру узла красно–черного–дерева. Он должен содержать следующие поля:

1. Key – поле для хранения ключа узла;
2. Data – поле для хранения данных узла;
3. Count – поле для хранения повторяющихся элементов в дереве;
4. IsRed – поле для хранения цвета узла.
5. Left – указатель на левого потомка.
6. Right – указатель на правого потомка.
7. Parent – указатель на предка.

Кроме полей, данный класс должен содержать также следующий метод:

1. Получение корня красно–черного дерева.

Структура, которую реализовывает класс **RBTreeNode** (см. рисунок 3.2):

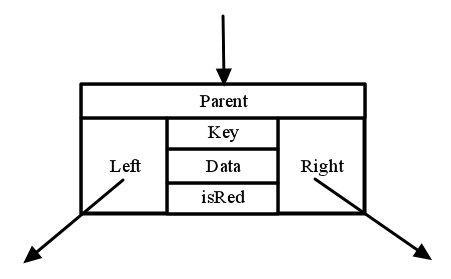


Рисунок 3.2 – Логическое представление структуры, которую реализовывает класс RBTreeNode

Класс **RBTree** должен представлять собой структуру красно–черного дерева. Он должен содержать следующие поля:

1. Root – поле для хранения указателя на корень дерева;

Кроме полей, данный класс должен содержать так же следующие методы.

1. Поворот поддерева влево.
2. Поворот поддерева вправо.
3. Поиск узла в дереве.
4. Вставка узла в дерево.
5. Удаление узла из дерева.

Структура, которую реализовывает данный класс (см. рисунок 3.3):

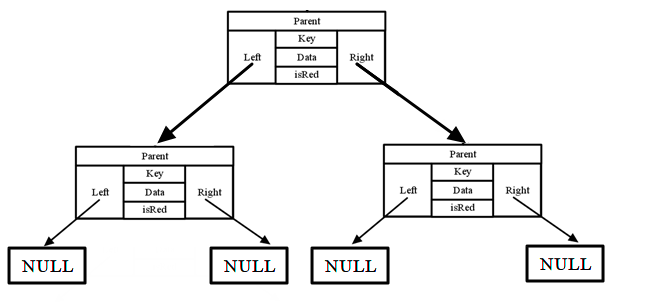


Рисунок 3.3 – Логическое представление структуры, которую реализовывает класс RBTree

# ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ

## Входные данные

### Входные данные

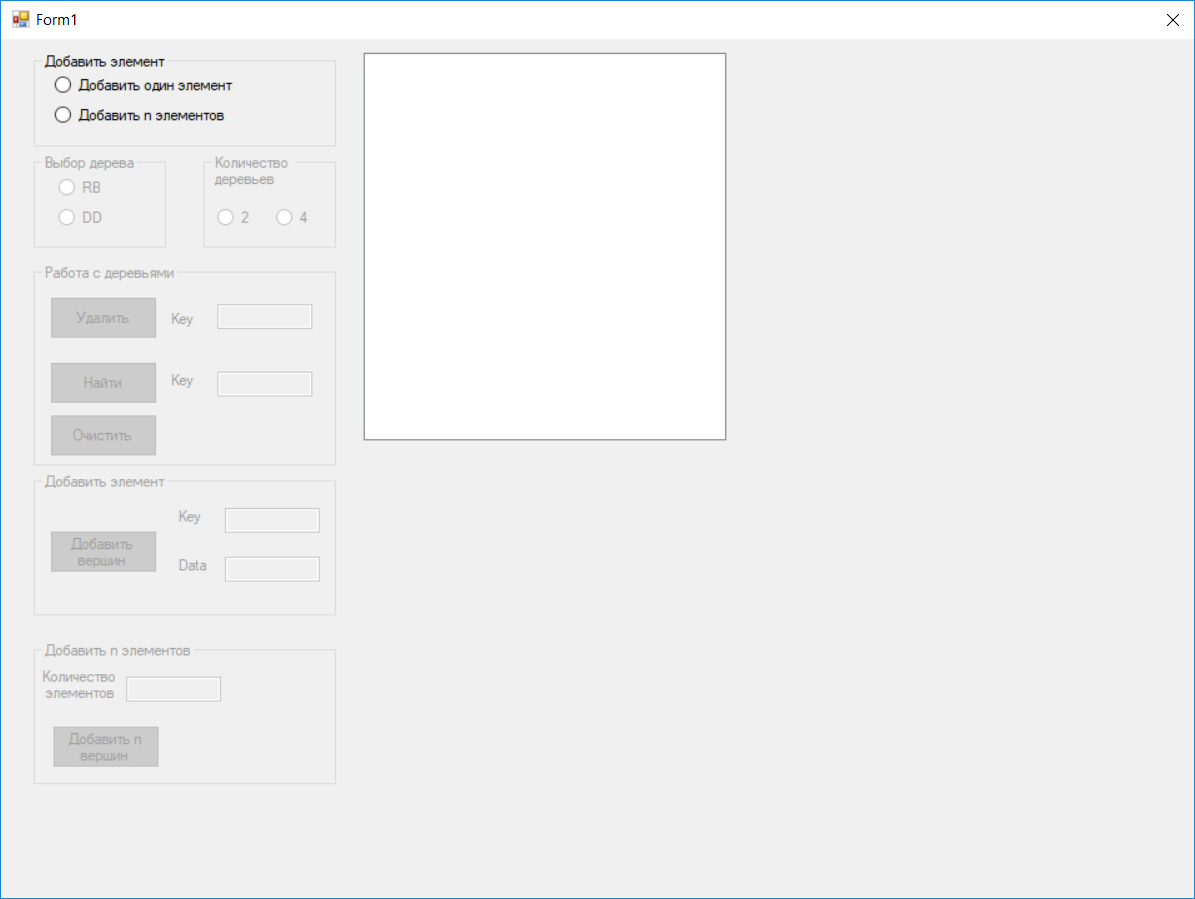
Входные данные приведены в таблице 4.1, входные формы изображены на рисунке 4.1:

Таблица 4.1 – Входные данные ПО

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Номер п/п | Наименование | Тип | Диапазон | Назначение |
| 1 | KEY | Целый | -2 147 483 648 <=KEY<=  2 147 483 647 | Ключ узла красно–черного дерева |
| 2 | DATA | Строковый | 0<= DATA<=65535 | Данные узла красно–черного дерева |
| 3 | radionbutton5 | Радиокнопка | 0<= N<=1 | Выбор количества элементов для добавления (по одному или по n сразу) |
| 4 | radionbutton6 | Радиокнопка | 2<= N<=4 | Выбор количества окон для отображения шагов добавления элементов дерева |

### Входные формы

На рисунке 4.1 изображена исходная экранная форма программы.



4

3

1

2

Рисунок 4.1 – Главная форма программы

Элементы формы, представленные на рисунке 4.1 имеют следующие назначения:

1. Кнопки для действий с деревом, добавление вершины дерева (одной или n штук), удаление, поиск и очистить (компоненты button);
2. Кнопки для выбора количества добавления элементов и количества отображаемых деревьев (компоненты radiobutton);
3. Запись ключа и данных в программу и надпись, поясняющая действие необходимое для выполнения (компоненты label и textbox);
4. Визуализация красно-черного дерева (компоненты treeview и picturebox).

## ВЫХОДНЫЕ ДАННЫЕ

### Выходные данные

На экран будут выведены красно-черные деревья. По ходу работы с программой будут выводится разные этапы добавления и/или удаления элементов красно-черного дерева. В результате ввода пользователю выведется либо сообщения об ошибке или выведется очередной этап работы с красно-черным деревом.

### Выходные формы

Выходными формами являются форма, в которой указан результат работы программы (изображение разных этапов добавления и/или удалени красно-черного дерева), окно сообщения, в котором указана информация об ошибке, в случае неверного ввода данных. Формы, для изображения графа.

Пример выходной формы, в случае успешной работы программы, изображен на рисунке 4.2

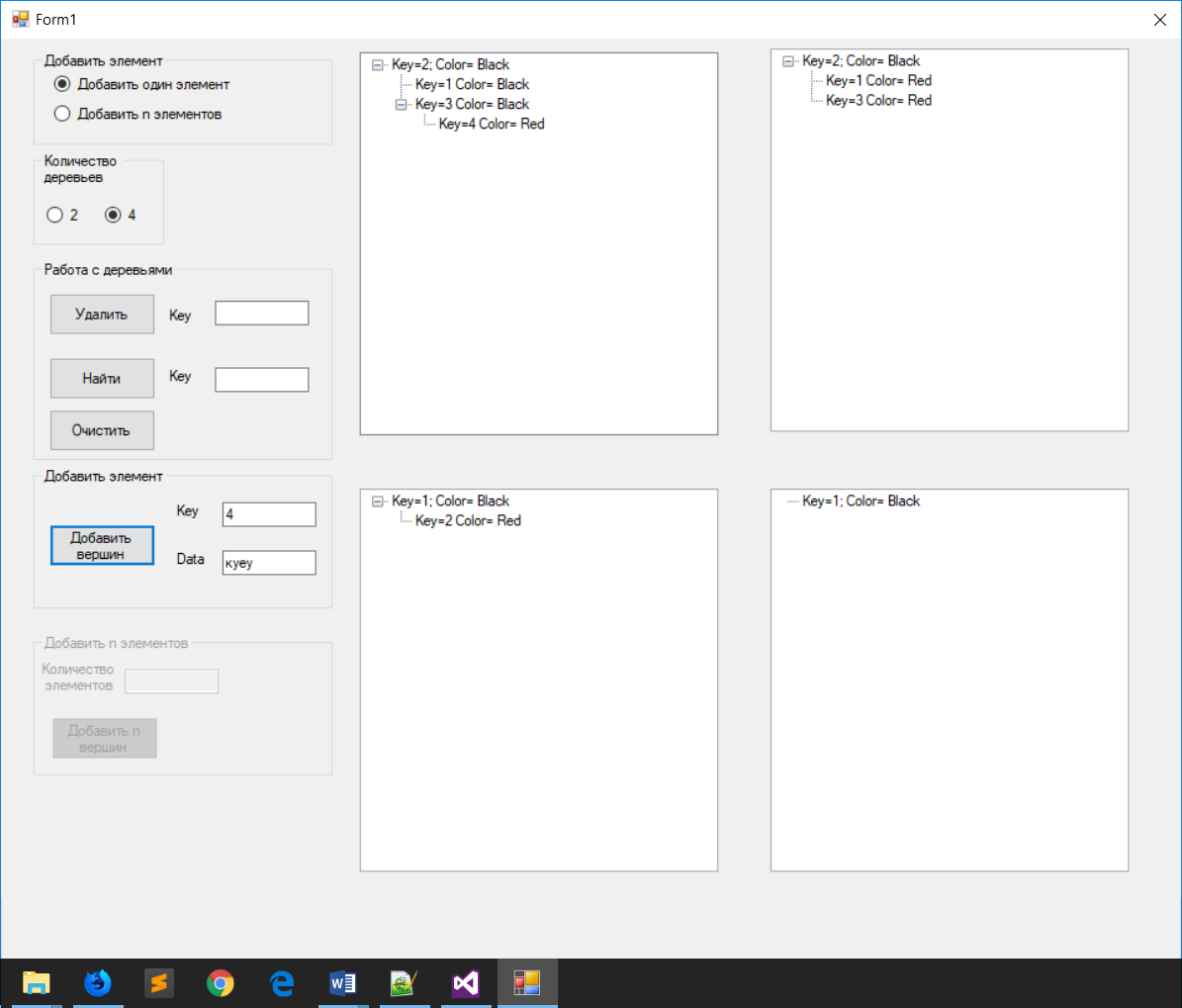


Рисунок 4.2 – Примеры результата расчетов

Пример выходной формы, в случае неправильного ввода данных, изображен на рисунке 4.3

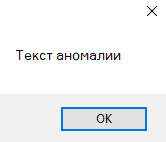


Рисунок 4.3 – Пример результата расчетов

Пример нормальной работы программы (см. рисунок 4.4)

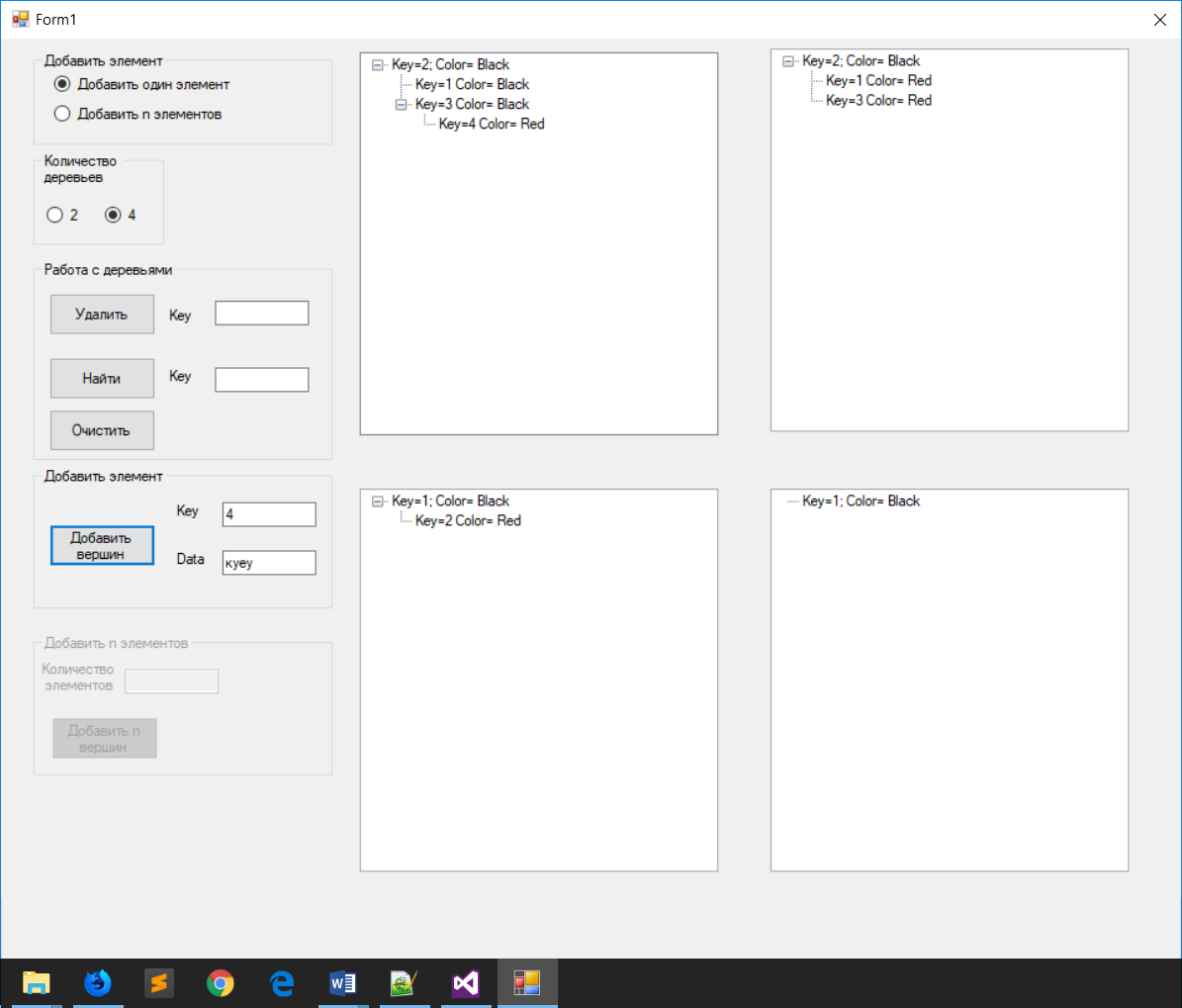


Рисунок 4.4 – Пример нормальной работы программы

Пример работы программы при аномальном результате   
(см. рисунок 4.5).

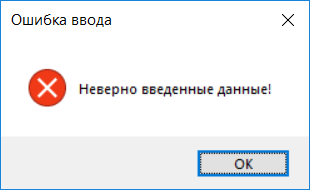
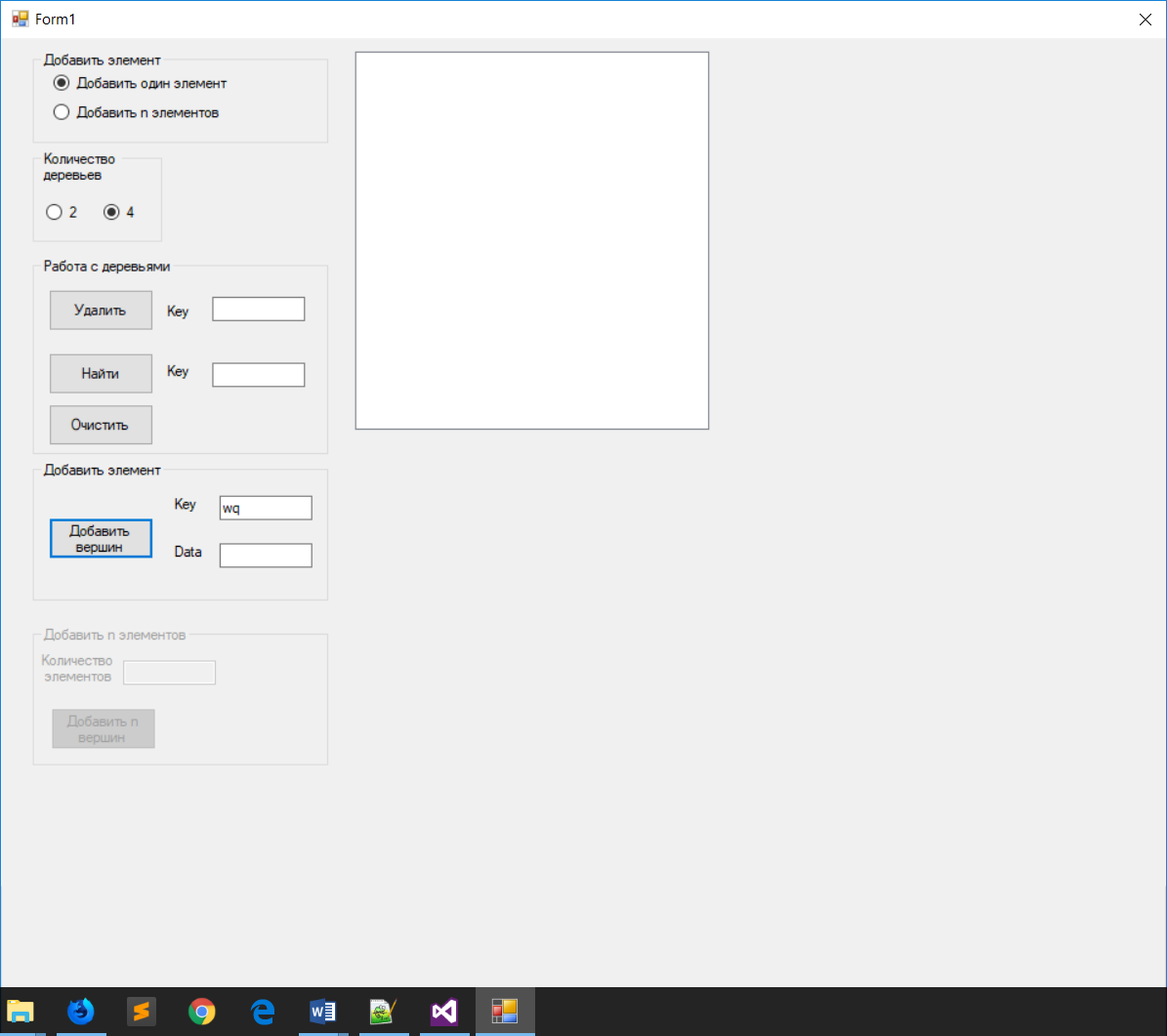


Рисунок 4.5 – Пример работы программы при аномальном результате

# АНОМАЛИИ

В данной программе аномалии в основном связаны с контролем входных данных. Они представлены в таблице 5.1.

Таблица 5.1 – Аномалии

|  |  |  |  |
| --- | --- | --- | --- |
| Но­мер п/п | Наименование | Условия  возникно­вения | Реакция |
| 1 | Не выбрано количество деревьев для показа | n = 2 или n = 4 | Вывод сообщения «Не выбрано количество деревьев для показа» |
| 2 | Неверно введен ключ для добавления | -2 147 483 648 <=key<=  2 147 483 647 | Вывод сообщения «Неверное введенные данные» |
| 3 | Неверно введены данные для добавления | 0<= data<=65535 | Вывод сообщения «Неверное введенные данные» |
| 4 | Пустой ввод | Ch = “ ” | Вывод сообщения «Неверное введенные данные» |
| *Примечание*   1. *key – ключ для добавления элемента красно-черного дерева;* 2. *data – данные для добавления элемента красно-черного дерева;* 3. *n – количество деревьев для показа;* 4. *Ch – символы для ввода.* | | | |

# ФУНКЦИОНАЛЬНЫЕ ТЕСТЫ

Была разработана программа на языке С#, листинг программы приведен в ПРИЛОЖЕНИ А. В таблице 6.1 показаны функциональные тесты в результате тестирования программы.

Таблица 6.1 – Функциональные тесты

|  |  |  |  |
| --- | --- | --- | --- |
| Номер п/п | Назначение | Входные данные | Результат |
| 1 | Проверка аномалии №1 | - | Вывод окна сообщения  «Не выбрано количество деревьев для показа» см. Рисунок 4.3. |
| 2 | Проверка аномалии №2 | ц | Вывод окна сообщения «Неверное введенные данные» см. Рисунок 4.3. |
| 3 | Проверка аномалии №3 | ц | Вывод окна сообщения  «Неверное введенные данные» см. Рисунок 4.3. |
| 4 | Проверка аномалии №4 | Отсутствие символов | Вывод окна сообщения  «Неверное введенные данные» см. Рисунок 4.3. |
| 8 | Добавление элемента в пустое дерево | Key – 1  Data – Тест1 | Результат добавления узла в пустое дерево с использованием предыдущих значений |
| 9 | Добавление элемента в дерево | Key – 2  Data – Тест2 | результат добавления узла в пустое дерево с использованием предыдущих значений |
| 10 | Генерация случайных 5 узлов и добавление в дерево | - | Программа правильно добавила 5 случайных узлов в существующее красно–черное дерево, т.к. были перекрашены остальные узлы с сохранением красно–черных свойств дерева |
| 11 | Удаление узла из красно–черного дерева | Тест2 | Программа правильно удалила узел из исходного красно–черного дерева, т.к. были перекрашены остальные узлы с сохранением красно–черных свойств дерева после удаления |

*Продолжение таблицы 6.1 – Функциональные тесты*

|  |  |  |  |
| --- | --- | --- | --- |
| 12 | Удаление несуществующего узла из красно–черного дерева | Тест2 | Программа правильно отреагировала на удаление несуществующего узла из красно–черного дерева, т.к. удаляемый узел действительно отсутствует в дереве |
| 13 | Поиск узла в красно–черном дереве | Key – 2 | Программа правильно сгенерировала информационное окно о найденном элементе красно–черного дерева, т.к. искомый элемент действительно присутствует в дереве |
| 14 | Поиск несуществующего узла в красно–черном дереве | Key – 55 | Программа правильно сгенерировала информационное окно оботсутствии элемента в красно–черном дереве, т.к. узел с таким ключом действительно отсутствует в дереве |
| 15 | Ввод слишком большого значения в поле для ключа | Key – 3e310 | Программа правильно сгенерировала окно об ошибке на ситуацию ввода слишком большого значения для ключа узла красно–черного дерева, так как вводимый параметр KEY содержал значение превышающее максимальную длину числового типа int |

В результате проведених функциональных тестов мы определили, какие данные будут показываться пользователю в результате ввода аномальных или корректных данных.

# АРХИТЕКТУРНОЕ ПРОЕКТИРОВАНИЕ ПО ПО ВИЗУАЛИЗАЦИИ МЕТОДОВ ВСТАВКИ, ПОИСКА И УДАЛЕНИЯ В КРАСНО – ЧЕРНЫХ ДЕРЕВЬЯХ

## Общая структура классов

Исходя из предыдущего пункта, в которой были спроектированы классы и взаимосвязи между ними, произведем архитектурное проектирование классов и их методов.

Таблица 7.1 – Структура классов проекта:

|  |  |
| --- | --- |
| RBTreeNode | // имя класса |
| поля |  |
| Key:TKey | // поле для хранения ключа узла |
| Data:TData | // поле для хранения данных узла |
| – Count:int | // поле для хранения количества дочерних элементов в дереве |
| – IsRed:bool | //поле для хранения цвета узла |
| – Left: RBTreeNode | //указатель на левого потомка |
| – Right: RBTreeNode | // указатель на правого потомка |
| – Parent: RBTreeNode | // указатель на предка |
| методы |  |
| + RBTreeNode(Key, Data) | // конструктор класса с параметрами |
| + Root () | // метод получения корня дерева |
| + Key () | // метод получения ключа узла; |
| + Data () | // метод получения данных узла |
| + Count () | // метод получения числа дочерних элементов в дереве |
| + IsRed () | // метод получения цвета узла |
| + Left () | // метод получения указателя на левого потомка |
| + Right () | // метод получения указателя на правого потомка |
| + Parent () | // метод получения указателя на предка |

*Продолжение таблицы 7.1 – Структура классов проекта:*

|  |  |
| --- | --- |
| RBTree | // имя класса |
| поля |  |
| Root:TNode | // поле для хранения ключа корня |
| методы |  |
| – RotateLeft(Node) | // метод поворота поддерева влево |
| – RotateRight (Node) | // метод поворота поддерева вправо |
| – Find(Key) | поиск узла в дереве по ключу |
| – Find(Key, Result) | // поиск узла в дереве по ключу с результатом Result |
| – InsertNode(Node) | // вставка узла в дерево |
| – DeleteNode(Node) | // удаление узла из дерева по узлу |
| – DeleteNode(Key) | // удаление узла из деревапо ключу |

Класс **RBTreeNode** является классом для описания узла красно–черного дерева и содержит метод получения корня дерева. Он реализовывает структуру одного узла. Класс содержит следующую информацию:

Поля класса:

* Key – поле для хранения ключа узла;
* Data – поле для хранения данных узла;
* Count –поле для хранения количества дочерних элементов в дереве;
* IsRed – поле для хранения цвета узла;
* Left – указатель на левого потомка
* Right – указатель на правого потомка
* Parent – указатель на предка

Методы класса:

* RBTreeNode(Key, Data) – конструктор класса с параметрами
* Root() – метод получения корня дерева
* Key() – метод получения ключа узла;
* Data() – метод получения данных узла;
* Count() –метод получения количества дочерних элементов в дереве;
* IsRed() – метод получения цвета узла;
* Left() – метод получения указателя на левого потомка
* Right() – метод получения указателя на правого потомка
* Parent() – метод получения указателя на предка

Класс **RBTree** реализовывают структуру красно–черного дерева, и содержит методы создания красно–черного дерева, добавления в него информации и получения с него искомой информации. Он содержит следующую информацию:

Поля класса:

* Root – указатель на корень дерева;

Методы класса:

* RotateLeft(Node) – метод поворота поддерева влево;
* RotateRight (Node) – метод поворота поддерева вправо;
* Find(Key) – поиск узла в дереве по ключу;
* Find(Key, Result) – поиск узла в дереве по ключу с результатом Result;
* InsertNode(Node) – вставка узла в дерево;
* DeleteNode(Node) – удаление узла из дерева по узлу;

## Подробное описание классов

### Класс RBTreeNode:

Шаблонный класс, который реализует структуру узла в красно–черном дереве.

Содержит следующие поля:

private bool \_IsRed – поле для хранения цвета узла;

private int \_Count – поле для хранения количества дочерних элементов в дереве;

private TData \_Data – поле для хранения данных узла;

private TKey \_Key – поле для хранения ключа узла;

private RBTreeNode<TKey, TData> \_Left – указатель на левого потомка;

private RBTreeNode<TKey, TData> \_Right – указатель на правого потомка;

private RBTreeNode<TKey, TData> \_Parent – указательнапредка узла;

class RBTreeNode<TKey, TData> where TKey : IComparable

{

private bool \_IsRed;

private int \_Count;

private TData \_Data;

private TKey \_Key;

private RBTreeNode<TKey, TData> \_Left;

private RBTreeNode<TKey, TData> \_Right;

private RBTreeNode<TKey, TData> \_Parent;

public RBTreeNode(TKey NewKey, TData NewData {… }

public bool IsRed {… }

public bool IsBlack {… }

public int Count {… }

public TKey Key {… }

public TData Data {… }

public RBTreeNode <TKey, TData> Left {… }

public RBTreeNode <TKey, TData> Right {… }

public RBTreeNode <TKey, TData> Parent {… }

public RBTreeNode <TKey, TData> Root {… }

}

Поля класса приведены в таблице 7.2.

## Таблица 7.2 – Поля и методы класса RBTreeNode

|  |  |
| --- | --- |
| RBTreeNode | //имя класса |
| Поля | |
| – \_IsRed:bool | // Логическое поле для хранения цвета узла дерева |
| – \_ Count:int | // Числовое поле для хранения количества дочерних элементов в дереве |
| – \_ Data:TData | // Поле шаблонного типа для хранения данных узла |
| – \_ Key:TKey | // Поле шаблонного типа для хранения ключа узла |
| – \_Left: RBTreeNode<TKey, TData> | // Указатель шаблонного типа на левого потомка дерева |
| – \_Right: RBTreeNode<TKey, TData> | // Указатель шаблонного типа на правого потомка дерева |
| – \_Parent: RBTreeNode<TKey, TData> | // Указатель шаблонного типа на предкаузла дерева |
| Методы | |
| + RBTreeNode(TKeyNewKey, TDataNewData) | // Конструктор класса RBTreeNode |
| + RBTreeNode <TKey,TData>Root | // Метод для получения корня дерева |
| + bool IsRed() | // Метод для получения данных поля \_IsRed. |
| + int Count() | // Метод для получения данных из поля \_Count. |
| + TKey Key() | // Метод для получения данных из поля \_Key. |
| + TData Data() | // Метод для получения данных из поля \_Data. |
| + RBTreeNode <TKey,TData> Left() | // Метод для получения данных из поля \_ Left. |
| + RBTreeNode <TKey,TData> Right() | // Метод для получения данных из поля \_Right. |

*Продолжение таблицы 7.2 – Поля и методы класса RBTreeNode*

|  |  |
| --- | --- |
| + RBTreeNode<TKey, TData> Parent() | // Метод для получения данных из поля \_Parent. |

### Класс RBTree

Класс, реализующий структуру красно–черного дерева. Содержит методы для создания красно–черного дерева, добавление узла и получения значения нужного узла. Структура класса приведена ниже:

class RBTree <TNode, TKey, TData>

where TNode : RBTreeNode<TKey, TData>

where TKey : IComparable

{

private TNode \_Root;

protected void RotateLeft(TNode Node) {… }

protected void RotateRight(TNode Node) {… }

public TNode Root {… }

public int Count {… }

public TNode Find(TKey Key, out int Result){… }

public TNode Find(TKey Key){… }

public void InsertNode(TNode Node){… }

public void DeleteNode(TNode Node) {… }

}

Поля класса приведены в таблице 7.2.

## Таблица 7.2 – Поля и методы класса RBTree

|  |  |
| --- | --- |
| InfElem | //имя класса |
| Поля |  |
| –\_Root:TNode | // Указатель шаблонного типа содержащая ссылку на корень дерева |
| Методы | |
| protected void RotateLeft (TNodeNode) | // Метод для вращения влево поддерева красно - черного дерева. |
| protected void RotateRight (TNodeNode) | // Метод для вращения вправо поддерева красно - черного дерева. |
| + TNodeFind(TKeyKey, out int Result) | // Метод для поиска узла в красно - черном дереве |

*Продолжение таблицы 7.2 ­– Поля и методы класса RBTree*

|  |  |
| --- | --- |
| + TNodeFind(TKeyKey) | // Метод для для проверки возвращаемого значения Result из метода Find(TKeyKey, out int Result) и возвращения соответствующего результата поиска: ссылку на найденный узел дерева либо null при отсутствии элемента в дереве. |
| + void InsertNode(TNodeNode) | // Метод для предназначен для добавления узла в дерево, подсчета совпадающих узлов с помощью метода Find(TKeyKey) для поиска узла дерева, а также метод восстанавливает красно–черные свойства дерева с помощью методов RotateLeft(TNodeNode) и RotateRight(TNodeNode), если в этом есть необходимость |
| + void DeleteNode(TNodeNode) | // Метод для предназначен для удаления узла из дерева, перерасчета совпадающих узлов, а также метод восстанавливает красно–черные свойства дерева после удаления с помощью методов RotateLeft(TNodeNode) и RotateRight(TNodeNode), если в этом есть необходимость. |

# НИСХОДЯЩЕЕ ПРОЕКТИРОВАНИЕ

Исходя из постановки задачи и спроектированных методов, проектируем алгоритмы этих методов.

## Метод InsertNode – метод для добавления узла в дерево:

void InsertNode(TNodeNode)

НАЧАЛО

int i;

RBTreeNode<TKey, TData> M, P, C;

ЕСЛИ ((Node == null) или (Node.Left != null) или (Node.Right != null) или (Node.Parent != null)) TO

ВОЗВРАТ;

КЕСЛИ

ЕСЛИ (\_Root == null) TO

\_Root = Node;

ВОЗВРАТ

КЕСЛИ

M = Find(Node.Key, out i);

ЕСЛИ (i == 0) TO

Node.Left = M.Left;

ЕСЛИ (Node.Left != null) TO

Node.Left.Parent = Node;

КЕСЛИ

Node.Right = M.Right;

ЕСЛИ (Node.Right != null) TO

Node.Right.Parent = Node;

КЕСЛИ

Node.Parent = M.Parent;

Node.IsRed = M.IsRed;

Node.Count = M.Count;

ЕСЛИ (M.Parent != null) TO

ЕСЛИ (M.Parent.Left == M)

M.Parent.Left = Node;

КЕСЛИ

ИНАЧЕ

M.Parent.Right = Node;

КЕСЛИ

if (\_Root == M) \_Root = Node;

M.Left = null;

M.Right = null;

M.Parent = null;

ВОЗВРАТ

КЕСЛИ

C = M;

ПОВТОР

C.Count++;

C = C.Parent;

ДО (C != null);

C = M;

Node.Parent = M;

Node.IsRed = true;

ЕСЛИ (i > 0) TO

M.Right = Node;

ИНАЧЕ

M.Left = Node;

КЕСЛИ

ПОКА ((Node.Parent != null) && (Node.Parent.IsRed)) ЦИКЛ

P = Node.Parent.Parent;

ЕСЛИ (Node.Parent == P.Left) TO

M = P.Right;

ЕСЛИ ((M != null) && (M.IsRed)) TO

Node.Parent.IsBlack = true;

M.IsBlack = true;

P.IsRed = true;

Node = (TNode)P

ИНАЧЕ

ЕСЛИ (Node == Node.Parent.Right) TO

Node = (TNode)Node.Parent;

RotateLeft(Node)

КЕСЛИ

Node.Parent.IsBlack = true;

P.IsRed = true;

RotateRight((TNode)P);

КЕСЛИ

ИНАЧЕ

M = P.Left;

ЕСЛИ ((M != null) && (M.IsRed)) TO

Node.Parent.IsBlack = true;

M.IsBlack = true;

P.IsRed = true;

Node = (TNode)P

ИНАЧЕ

ЕСЛИ (Node == Node.Parent.Left) TO

Node = (TNode)Node.Parent;

RotateRight(Node)

КЕСЛИ

Node.Parent.IsBlack = true;

P.IsRed = true;

RotateLeft((TNode)P);

КЕСЛИ

КЕСЛИ

КЦИКЛ

C = C.Root;

C.IsBlack = true;

\_Root = (TNode)C;

КОНЕЦ

## Метод DeleteNode – метод удаления узла из красно–черного дерева.

void DeleteNode(TNode Node)

НАЧАЛО

int i;

RBTreeNode<TKey, TData> M, P, C, T;

ЕСЛИ (Node.Root != \_Root) TO

ВОЗВРАТ;

КЕСЛИ

M = Node;

ЕСЛИ (M.Left == null) TO

C = M.Right;

ИНАЧЕ

ЕСЛИ (M.Right == null)

C = M.Left;

КЕСЛИ

КЕСЛИ

ЕСЛИ

M = M.Right;

ПОКА (M.Left != null) ЦИКЛ

M = M.Left;

КЦИКЛ

C = M.Right;

КЕСЛИ

ЕСЛИ (M != Node) TO

M.Count = Node.Count – 1;

T = M.Parent;

ПОКА (T != null) ЦИКЛ

T.Count––;

T = T.Parent;

КЦИКЛ

Node.Left.Parent = M;

M.Left = Node.Left;

ЕСЛИ (M != Node.Right) TO

P = M.Parent;

ЕСЛИ (C != null) TO

C.Parent = P;

КЕСЛИ

P.Left = C;

M.Right = Node.Right;

Node.Right.Parent = M

ИНАЧЕ

P = M;

КЕСЛИ

ЕСЛИ (Node.Parent != null) TO

ЕСЛИ (Node.Parent.Left == Node)

Node.Parent.Left = M;

КЕСЛИ

ИНАЧЕ

Node.Parent.Right = M;

КЕСЛИ

M.Parent = Node.Parent;

i = (M.IsRed ? 1 : 0);

M.IsRed = Node.IsRed;

Node.IsRed = (i != 0)

ИНАЧЕ

T = Node.Parent;

ПОКА (T != null) ЦИКЛ

T.Count––;

T = T.Parent;

КЦИКЛ

P = M.Parent;

ЕСЛИ (C != null) TO

C.Parent = P;

КЕСЛИ

ЕСЛИ (Node.Parent != null) TO

if (Node.Parent.Left == Node) Node.Parent.Left = C;

ИНАЧЕ

Node.Parent.Right = C;

КЕСЛИ

i = (M.IsRed ? 1 : 0);

КЕСЛИ

ЕСЛИ ((M == Node) и (M.Left == null) и (M.Right == null) и (M.Parent == null)) TO

\_Root = null;

ВОЗВРАТ

КЕСЛИ

Node.Left = null;

Node.Right = null;

Node.Parent = null;

ЕСЛИ (P == null) TO

Node = (TNode)C;

ИНАЧЕ

Node = (TNode)P;

КЕСЛИ

ЕСЛИ (i == 0) TO

ПОКА ((P != null) && ((C == null) или (C.IsBlack))) ЦИКЛ

ЕСЛИ (C == P.Left) TO

M = P.Right;

ЕСЛИ (M.IsRed) TO

M.IsBlack = true;

P.IsRed = true;

RotateLeft((TNode)P);

M = P.Right

КЕСЛИ

ЕСЛИ (((M.Left == null) или (M.Left.IsBlack)) && ((M.Right == null) или (M.Right.IsBlack))) TO

M.IsRed = true;

C = P;

P = P.Parent

ИНАЧЕ

ЕСЛИ ((M.Right == null) или (M.Right.IsBlack)) TO

M.Left.IsBlack = true;

M.IsRed = true;

RotateRight((TNode)M);

M = P.Right

КЕСЛИ

M.IsRed = P.IsRed;

P.IsBlack = true;

ЕСЛИ (M.Right != null) TO

M.Right.IsBlack = true;

КЕСЛИ

RotateLeft((TNode)P);

break;

КЕСЛИ

ИНАЧЕ

M = P.Left;

ЕСЛИ (M.IsRed) TO

M.IsBlack = true;

P.IsRed = true;

RotateLeft((TNode)P);

M = P.Left

КЕСЛИ

ЕСЛИ (((M.Right == null) или (M.Right.IsBlack)) && ((M.Left == null) или (M.Left.IsBlack))) TO

M.IsRed = true;

C = P;

P = P.Parent

ИНАЧЕ

ЕСЛИ ((M.Left == null) или (M.Left.IsBlack)) TO

M.Right.IsBlack = true;

M.IsRed = true;

RotateLeft((TNode)M);

M = P.Left

КЕСЛИ

M.IsRed = P.IsRed;

P.IsBlack = true;

ЕСЛИ (M.Left != null) TO

M.Left.IsBlack = true;

КЕСЛИ

RotateRight((TNode)P);

break;

КЕСЛИ

КЕСЛИ

КЦИКЛ

ЕСЛИ (C != null) TO

C.IsBlack = true;

КЕСЛИ

КЕСЛИ

\_Root = (TNode)Node.Root;

КОНЕЦ

## Метод Find – метод для поиска узла в красно–черном дерева.

TNode Find(TKey Key, out int Result)

НАЧАЛО

int i;

RBTreeNode<TKey, TData> N;

ЕСЛИ (\_Root == null) TO

Result = 1;

ВОЗВРАТ null

КЕСЛИ

N = \_Root;

ПОКА (true) ЦИКЛ

i = Key.CompareTo(N.Key);

ЕСЛИ (i > 0) TO

ЕСЛИ (N.Right == null) TO

Result = 1;

ВОЗВРАТ (TNode)N

КЕСЛИ

N = N.Right

ИНАЧЕ

ЕСЛИ (i< 0)

ЕСЛИ (N.Left == null)

Result = –1;

КЕСЛИ

ВОЗВРАТ (TNode)N;

КЕСЛИ

N = N.Left;

КЕСЛИ

ЕСЛИ

Result = 0;

ВОЗВРАТ (TNode)N;

КЕСЛИ

КЦИКЛ

КОНЕЦ

## Метод RotateRight – метод для вращения поддерева вправо.

void RotateRight(TNode Node)

НАЧАЛО

RBTreeNode<TKey, TData> P, N;

ЕСЛИ ((Node == null) или (Node.Left == null)) TO

ВОЗВРАТ;

КЕСЛИ

P = Node.Parent;

N = Node.Left;

ЕСЛИ (P != null) TO

ЕСЛИ (P.Right == Node)

P.Right = N;

КЕСЛИ

ИНАЧЕ

P.Left = N;

КЕСЛИ

Node.Left = N.Right;

ЕСЛИ (Node.Left != null) TO

Node.Left.Parent = Node;

КЕСЛИ

N.Parent = P;

Node.Parent = N;

N.Right = Node;

Node.Count = 1;

ЕСЛИ (Node.Left != null) TO

Node.Count += Node.Left.Count;

КЕСЛИ

ЕСЛИ (Node.Right != null) TO

Node.Count += Node.Right.Count;

КЕСЛИ

N.Count = 1;

ЕСЛИ (N.Left != null) TO

N.Count += N.Left.Count;

КЕСЛИ

ЕСЛИ (N.Right != null) TO

N.Count += N.Right.Count;

КЕСЛИ

КОНЕЦ

Метод RotateLeft аналогичен методу RotateRight.

Ниже, на рисунках 8.1 – 8.2 представленны UML-диаграммы для Class.cs. и для Form1.cs.

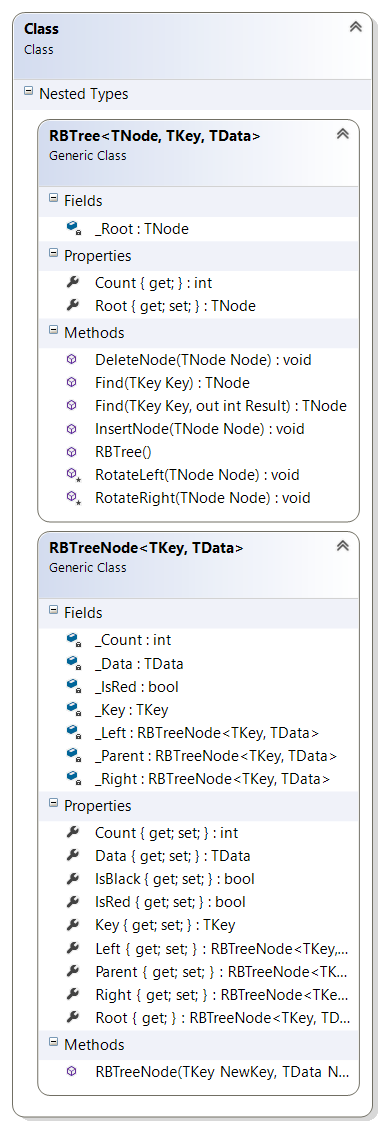


Рисунок 9.1 – UML-диаграмма класса Class

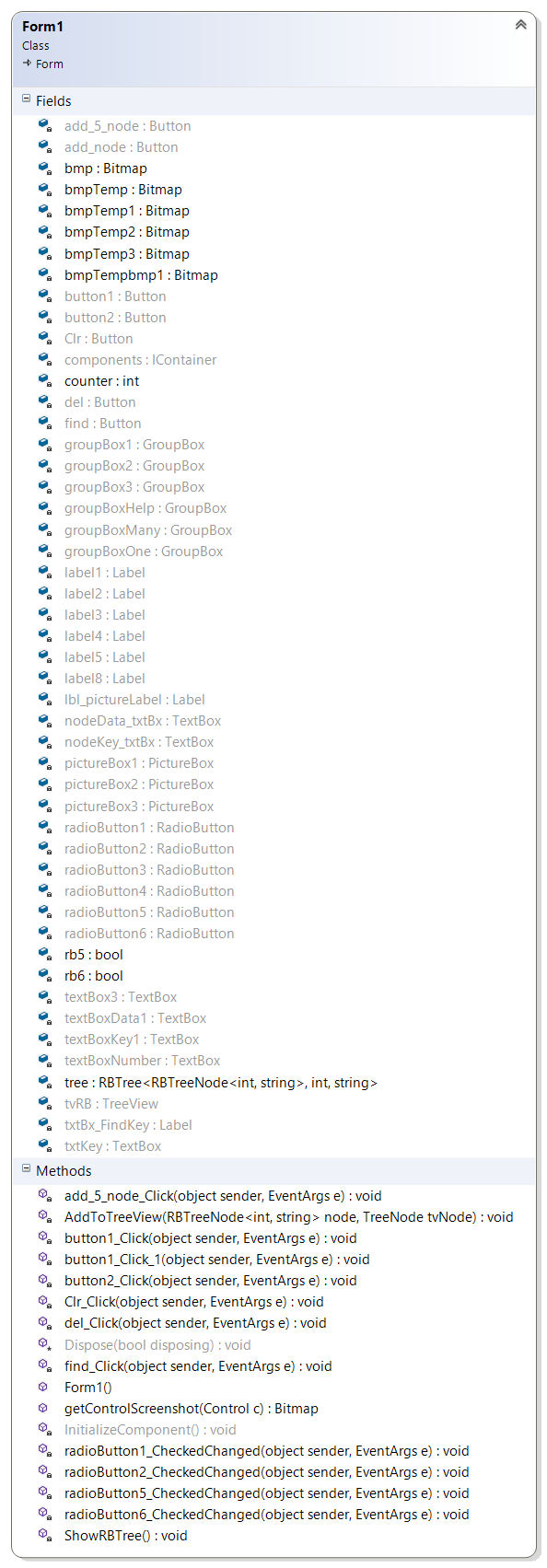


Рисунок 9.2 – UML-диаграмма класса Form1

# ЛОГИЧЕСКАЯ МОДЕЛЬ ПРОГРАММЫ

Схема всех подзадач и задач программного проекта представлена на рисунке 10.1.

|  |  |
| --- | --- |
|  | Рисунок 10.1 – Логическая модель программы |

Листинг кода программы, реализованной на языке программирования C#, приведен в *ПРИЛОЖЕНИИ А*, а экранные формы – в *ПРИЛОЖЕНИИ Б.* Слайды презентации приведены в *ПРИЛОЖЕНИИ В.*

# ВЫВОД

Тестирование показало, что ПО корректно работает с реализованным алгоритмом красно–черного дерева. Реализация ПО включает ввод и контроль данных с экранной формы, что значительно улучшает устойчивость программы к сбоям.

Тестирование всех возможных реакций программы на входные данные показало, что разработанное ПО соответствует требованиям постановке задачи.

В ходе архитектурного проектирования были разработаны и описаны классы: RBTreeNode, RBTree. Логическая модель структуры проекта представлена на UML – диаграмме (приложение Б). В ней изображены 2 класса RBTreeNode, RBTree – реализующие алгоритм «красно–черное дерево».

Также была разработана физическая модель ПО. Для создания физической модели ПО были изучены связи между файлами, которые входят в состав проекта (приложение В). Для создания такой модели на основе проекта С# были рассмотрены зависимости в стандартном пространстве имен System файлов с расширением \*.cs.

В результате проектирования были получены алгоритмы добавления, удаления, поиска узла в красно–черном дереве, а также повороты поддерева вправо и влево. Программный код ПО реализован на языке С#.

В ходе выполнения курсового проекта было:

* Изучено структуру данных самосбалансирующегося красно–черного дерева поиска.
* Составлены метод решения алгоритма «красно–черного дерева», где было определено:

1. Свойства «красно - черного дерева».
2. Операции «красно - черного дерева».
3. Вставка элемента в «красно - черное дерево».
4. Удаление элемента из «красно - черного дерева».
5. Поиск узла в «красно - черном дереве».
6. Преимущества «красно - черного дерева».

* Смоделированы два базовых класса:

1. RBTreeNode – описывающий узел «красно–черного дерева».
2. RBTree – содержащий структуру и методы взаимодействия с красно - черным деревом.

* Определены входные и выходные данные, входные и выходные формы ПО «Визуализатор красно - черного дерева»
* Составлена архитектура классов полученных в результате моделирования.
* Спроектирована реализация ПО«Визуализатор красно - черного дерева»
* Проведено тестирование ПО «Визуализатор красно - черного дерева». Тестирование всех возможных реакций программы на входные данные показало, что разработанное ПО соответствует требованиям постановки задачи.

На основе проделанной работы было создано ПО для визуализации методов вставки, поиска и удаления в красно - черных деревьях.

# ПЕРЕЧЕНЬ ССЫЛОК

1. Алгоритмы: построение и анализ– 2–е изд. – М.: Издательский дом «Вильямс», Кормен Т., Лейзерсон Ч., Ривест Р., Штайн К. 2011
2. «Структуры данных и алгоритмы» А.В. Ахо, Д.Э.Хопкрофт, Д.Д.Ульман: 2010.
3. Chris Okasaki. Red–black trees in a functional setting. J. Funct. Program., 9(4):471–477, 1999.
4. Основы программирования и алгоритмические языки: учеб. пособие по курсовому проектированию / Н. Г. Мокляк, Е. В. Соколова, В. В. Дужая. – Х.: Нац. аэрокосм. ун–т им. Н. Е. Жуковского «Харьк. авиац. ин–т», 2012. – 62 с.
5. [Алгоритмы: вводный курс](http://www.williamspublishing.com/Books/978-5-8459-1868-0.html), Томас Х. Кормен
6. [Алгоритмические трюки для программистов, 2–е издание](http://www.williamspublishing.com/Books/978-5-8459-1838-3.html), Генри С. Уоррен, 2008г. – 405с.
7. [Алгоритмы: построение и анализ, 3–е издание](http://www.williamspublishing.com/Books/978-5-8459-1794-2.html), Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн, 2006г. – 315с.
8. [Алгоритмы на Java, 4–е издание](http://www.williamspublishing.com/Books/978-5-8459-1781-2.html), Роберт Седжвик, Кевин Уэйн, 2010г. – 203с.
9. [Искусство программирования, том 4А. Комбинаторные алгоритмы , часть 1](http://www.williamspublishing.com/Books/978-5-8459-1744-7.html), Дональд Эрвин Кнут, 2008г. – 244с.
10. [Алгоритмы на C++. Фундаментальные алгоритмы и структуры данных. 2 книги в одной !](http://www.williamspublishing.com/Books/978-5-8459-1650-1.html), Роберт Седжвик, 2007г. – 455с.
11. [Программирование: принципы и практика использования C++, исправленное издание](http://www.williamspublishing.com/Books/978-5-8459-1705-8.html), Бьярне Страуструп, 2006г. – 145с.
12. [Алгоритмы: построение и анализ, 2–е издание](http://www.williamspublishing.com/Books/5-8459-0857-4.html), Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн, 2011г. – 44с.
13. [Искусство программирования, том 1. Основные алгоритмы, 3–е издание](http://www.williamspublishing.com/Books/sci_Knuth1.html), Дональд Э. Кнут, 2011г. – 515с.
14. «Алгоритмы и структуры данных», Никлаус Вирт, 2010. – 105с.

# *ПРИЛОЖЕНИЕ А*

Листинг программы

**Файл Class.cs**

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

namespace course

{

public class Class

{

/// <summary>

/// Класс описывающий элемент узла красно-черного дерева

/// </summary>

/// <typeparam name="TKey"></typeparam>

/// <typeparam name="TData"></typeparam>

public class RBTreeNode<TKey, TData> where TKey : IComparable

{

private bool \_IsRed;

private int \_Count;

private TData \_Data;

private TKey \_Key;

private RBTreeNode<TKey, TData> \_Left;

private RBTreeNode<TKey, TData> \_Right;

private RBTreeNode<TKey, TData> \_Parent;

public RBTreeNode(TKey NewKey, TData NewData)

{

\_IsRed = false;

\_Count = 1;

\_Data = NewData;

\_Key = NewKey;

\_Left = null;

\_Right = null;

\_Parent = null;

}

public bool IsRed { get { return \_IsRed; } set { \_IsRed = value; } }

public bool IsBlack { get { return !\_IsRed; } set { \_IsRed = !value; } }

public int Count { get { return \_Count; } set { \_Count = value; } }

public TKey Key { get { return \_Key; } set { \_Key = value; } }

public TData Data { get { return \_Data; } set { \_Data = value; } }

public RBTreeNode<TKey, TData> Left { get { return \_Left; } set { \_Left = value; } }

public RBTreeNode<TKey, TData> Right { get { return \_Right; } set { \_Right = value; } }

public RBTreeNode<TKey, TData> Parent { get { return \_Parent; } set { \_Parent = value; } }

//Метод получения корня дерева

public RBTreeNode<TKey, TData> Root

{

get

{

RBTreeNode<TKey, TData> Node;

Node = this;

while (Node.\_Parent != null) Node = Node.\_Parent;

return Node;

}

}

}

/// <summary>

/// Класс описывающий красно-черное дерево

/// </summary>

/// <typeparam name="TNode"></typeparam>

/// <typeparam name="TKey"></typeparam>

/// <typeparam name="TData"></typeparam>

public class RBTree<TNode, TKey, TData>

where TNode : RBTreeNode<TKey, TData>

where TKey : IComparable

{

private TNode \_Root;

public RBTree()

{

\_Root = null;

}

//Поворот влево

protected void RotateLeft(TNode Node)

{

RBTreeNode<TKey, TData> P, N;

if ((Node == null) || (Node.Right == null)) return;

P = Node.Parent;

N = Node.Right;

if (P != null)

if (P.Left == Node) P.Left = N; else P.Right = N;

Node.Right = N.Left;

if (Node.Right != null) Node.Right.Parent = Node;

N.Parent = P;

Node.Parent = N;

N.Left = Node;

//корректировка count

Node.Count = 1;

if (Node.Left != null) Node.Count += Node.Left.Count;

if (Node.Right != null) Node.Count += Node.Right.Count;

N.Count = 1;

if (N.Left != null) N.Count += N.Left.Count;

if (N.Right != null) N.Count += N.Right.Count;

}

//Поворот вправо

protected void RotateRight(TNode Node)

{

RBTreeNode<TKey, TData> P, N;

if ((Node == null) || (Node.Left == null)) return;

P = Node.Parent;

N = Node.Left;

if (P != null)

if (P.Right == Node) P.Right = N; else P.Left = N;

Node.Left = N.Right;

if (Node.Left != null) Node.Left.Parent = Node;

N.Parent = P;

Node.Parent = N;

N.Right = Node;

//корректировка count

Node.Count = 1;

if (Node.Left != null) Node.Count += Node.Left.Count;

if (Node.Right != null) Node.Count += Node.Right.Count;

N.Count = 1;

if (N.Left != null) N.Count += N.Left.Count;

if (N.Right != null) N.Count += N.Right.Count;

}

public TNode Root

{

get { return \_Root; }

set { if (value == null) \_Root = null; }

}

public int Count

{

get

{

if (\_Root == null) return 0;

else return \_Root.Count;

}

}

//Вспомогательный метод для поиска узла

public TNode Find(TKey Key, out int Result)

{

int i;

RBTreeNode<TKey, TData> N;

if (\_Root == null)

{

Result = 1;

return null;

}

N = \_Root;

while (true)

{

i = Key.CompareTo(N.Key);

if (i > 0)

{

if (N.Right == null)

{

Result = 1;

return (TNode)N;

}

N = N.Right;

}

else if (i < 0)

{

if (N.Left == null)

{

Result = -1;

return (TNode)N;

}

N = N.Left;

}

else

{

Result = 0;

return (TNode)N;

}

}

}

//Поиск узла в дереве

public TNode Find(TKey Key)

{

int i;

RBTreeNode<TKey, TData> N;

N = Find(Key, out i);

if (i != 0) N = null;

return (TNode)N;

}

//Вставка узла в дерево

public void InsertNode(TNode Node)

{

int i;

/\*

\* M - удаляемы узел

\* Р - новый предок

\* С - потомок М

\*/

RBTreeNode<TKey, TData> M, P, C;

if ((Node == null) || (Node.Left != null) || (Node.Right != null) || (Node.Parent != null)) return;

if (\_Root == null)

{

\_Root = Node;

return;

}

M = Find(Node.Key, out i);

if (i == 0) // заменяем узел

{

Node.Left = M.Left;

if (Node.Left != null) Node.Left.Parent = Node;

Node.Right = M.Right;

if (Node.Right != null) Node.Right.Parent = Node;

Node.Parent = M.Parent;

Node.IsRed = M.IsRed;

Node.Count = M.Count;

if (M.Parent != null)

if (M.Parent.Left == M) M.Parent.Left = Node; else M.Parent.Right = Node;

if (\_Root == M) \_Root = Node;

M.Left = null;

M.Right = null;

M.Parent = null;

return;

}

//Перерасчитываем кол-во узлов Count

C = M;

do

{

C.Count++;

C = C.Parent;

} while (C != null);

C = M;

Node.Parent = M;

Node.IsRed = true;

if (i > 0) M.Right = Node; else M.Left = Node;

//перекраска RBTree после вставки

while ((Node.Parent != null) && (Node.Parent.IsRed))

{

P = Node.Parent.Parent;

if (Node.Parent == P.Left)

{

M = P.Right;

if ((M != null) && (M.IsRed))

{

Node.Parent.IsBlack = true;

M.IsBlack = true;

P.IsRed = true;

Node = (TNode)P;

}

else

{

if (Node == Node.Parent.Right)

{

Node = (TNode)Node.Parent;

RotateLeft(Node);

}

Node.Parent.IsBlack = true;

P.IsRed = true;

RotateRight((TNode)P);

}

}

else

{

M = P.Left;

if ((M != null) && (M.IsRed))

{

Node.Parent.IsBlack = true;

M.IsBlack = true;

P.IsRed = true;

Node = (TNode)P;

}

else

{

if (Node == Node.Parent.Left)

{

Node = (TNode)Node.Parent;

RotateRight(Node);

}

Node.Parent.IsBlack = true;

P.IsRed = true;

RotateLeft((TNode)P);

}

}

}

C = C.Root;

C.IsBlack = true;

\_Root = (TNode)C;

}

//Удаление узла из дерева

public void DeleteNode(TNode Node)

{

int i;

RBTreeNode<TKey, TData> M, P, C, T;

/\*

\* M - удаляемы узел

\* Т - новый предок

\* Р - дополнительная ссылка на предок

\* С - потомок М

\*/

if (Node.Root != \_Root) return;

M = Node;

if (M.Left == null) C = M.Right;

else if (M.Right == null) C = M.Left;

else

{

M = M.Right;

while (M.Left != null) M = M.Left;

C = M.Right;

}

if (M != Node)

{

//Корректировка Count

M.Count = Node.Count - 1;

T = M.Parent;

while (T != null)

{

T.Count--;

T = T.Parent;

}

Node.Left.Parent = M;

M.Left = Node.Left;

if (M != Node.Right)

{

P = M.Parent;

if (C != null) C.Parent = P;

P.Left = C;

M.Right = Node.Right;

Node.Right.Parent = M;

}

else P = M;

if (Node.Parent != null)

if (Node.Parent.Left == Node) Node.Parent.Left = M; else Node.Parent.Right = M;

M.Parent = Node.Parent;

i = (M.IsRed ? 1 : 0);

M.IsRed = Node.IsRed;

Node.IsRed = (i != 0);

}

else

{

//Корректировка Count

T = Node.Parent;

while (T != null)

{

T.Count--;

T = T.Parent;

}

P = M.Parent;

if (C != null) C.Parent = P;

if (Node.Parent != null)

if (Node.Parent.Left == Node) Node.Parent.Left = C; else Node.Parent.Right = C;

i = (M.IsRed ? 1 : 0);

}

if ((M == Node) && (M.Left == null) && (M.Right == null) && (M.Parent == null))

{

\_Root = null;

return;

}

Node.Left = null;

Node.Right = null;

Node.Parent = null;

if (P == null) Node = (TNode)C; else Node = (TNode)P;

//перекраска RBTree после удаления

if (i == 0) //черный

{

while ((P != null) && ((C == null) || (C.IsBlack)))

{

if (C == P.Left)

{

M = P.Right;

if (M.IsRed)

{

M.IsBlack = true;

P.IsRed = true;

RotateLeft((TNode)P);

M = P.Right;

}

if (((M.Left == null) || (M.Left.IsBlack)) && ((M.Right == null) || (M.Right.IsBlack)))

{

M.IsRed = true;

C = P;

P = P.Parent;

}

else

{

if ((M.Right == null) || (M.Right.IsBlack))

{

M.Left.IsBlack = true;

M.IsRed = true;

RotateRight((TNode)M);

M = P.Right;

}

M.IsRed = P.IsRed;

P.IsBlack = true;

if (M.Right != null) M.Right.IsBlack = true;

RotateLeft((TNode)P);

break;

}

}

else //если C=P.Right

{

M = P.Left;

if (M.IsRed)

{

M.IsBlack = true;

P.IsRed = true;

RotateLeft((TNode)P);

M = P.Left;

}

if (((M.Right == null) || (M.Right.IsBlack)) && ((M.Left == null) || (M.Left.IsBlack)))

{

M.IsRed = true;

C = P;

P = P.Parent;

}

else

{

if ((M.Left == null) || (M.Left.IsBlack))

{

M.Right.IsBlack = true;

M.IsRed = true;

RotateLeft((TNode)M);

M = P.Left;

}

M.IsRed = P.IsRed;

P.IsBlack = true;

if (M.Left != null) M.Left.IsBlack = true;

RotateRight((TNode)P);

break;

}

}

}

if (C != null) C.IsBlack = true;//черный

}

\_Root = (TNode)Node.Root;

} } }}

**ФайлForm1.cs**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using System.Windows.Forms;

namespace course

{

public partial class Form1 : Form

{

private Class.RBTree<Class.RBTreeNode<int, string>, int, string> tree;

public Form1()

{

InitializeComponent();

tree = new Class.RBTree<Class.RBTreeNode<int, string>, int, string>();

}

Bitmap bmpTemp = null;

Bitmap bmp = null;

Bitmap bmpTemp1 = null;

Bitmap bmpTemp2 = null;

Bitmap bmpTemp3 = null;

Bitmap bmpTempbmp1 = null;

bool rb5 = false, rb6 = false;

public Bitmap getControlScreenshot(Control c)

{

Bitmap res = new Bitmap(c.Width, c.Height);

c.DrawToBitmap(res, new Rectangle(Point.Empty, c.Size));

return res;

}

private void AddToTreeView(Class.RBTreeNode<int, string> node, TreeNode tvNode)

{

if (node == null) return;

TreeNode newtvNode;

if (node.IsBlack)

{

newtvNode = new TreeNode("Key=" + node.Key.ToString() + " Color= Black");

tvNode.Nodes.Add(newtvNode);

AddToTreeView(node.Left, newtvNode);

AddToTreeView(node.Right, newtvNode);

}

else

{

newtvNode = new TreeNode("Key=" + node.Key.ToString() + " Color= Red");

tvNode.Nodes.Add(newtvNode);

AddToTreeView(node.Left, newtvNode);

AddToTreeView(node.Right, newtvNode);

}

// i = (node.IsBlack ? 0 : 1);

//newtvNode = new TreeNode("Key=" + node.Key.ToString() + "; Data=" + node.Data.ToString() + "; Count=" + node.Count, i, i);

//tvNode.Nodes.Add(newtvNode);

//AddToTreeView(node.Left, newtvNode);

//AddToTreeView(node.Right, newtvNode);

}

private void ShowRBTree()

{

try

{

tvRB.Nodes.Clear();

if (tree.Count == 0) return;

string color;

if (tree.Root.IsBlack)

{

color = "black";

tvRB.Nodes.Add(tree.Root.Key.ToString(), "Key=" + tree.Root.Key.ToString() + "; Color= Black");

AddToTreeView(tree.Root.Left, tvRB.Nodes[0]);

AddToTreeView(tree.Root.Right, tvRB.Nodes[0]);

}

else

{

color = "red";

tvRB.Nodes.Add(tree.Root.Key.ToString(), "Key=" + tree.Root.Key.ToString() + "; Color= Red");

AddToTreeView(tree.Root.Left, tvRB.Nodes[0]);

AddToTreeView(tree.Root.Right, tvRB.Nodes[0]);

}

//tvRB.Nodes.Add(tree.Root.Key.ToString(), "Key=" + tree.Root.Key.ToString() + "; Data=" + tree.Root.Data.ToString() + "; Count=" + tree.Root.Count, (tree.Root.IsBlack ? 0 : 1));

//tvRB.Nodes.Add(tree.Root.Key.ToString(), "Key=" + tree.Root.Key.ToString() + "; Data=" + tree.Root.Data.ToString() + "; Count=" + tree.Root.Count, (tree.Root.IsBlack ? 0 : 1));

//AddToTreeView(tree.Root.Left, tvRB.Nodes[0]);

//AddToTreeView(tree.Root.Right, tvRB.Nodes[0]);

tvRB.ExpandAll();

if (bmpTemp != null)

{

lbl\_pictureLabel.Visible = true;

}

}

catch (Exception)

{

MessageBox.Show("Неверно введенные данные!", "Ошибка ввода", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

private void button1\_Click(object sender, EventArgs e)

{

Class.RBTreeNode<int, string> node;

int key;

string data;

//for view picture

try

{

key = Convert.ToInt32(nodeKey\_txtBx.Text);

data = nodeData\_txtBx.Text.ToString();

node = new Class.RBTreeNode<int, string>(key, data);

tree.InsertNode(node);

bmpTemp = bmp;

bmpTemp1 = bmpTemp;

bmpTemp2 = bmpTemp1;

ShowRBTree();

bmp = getControlScreenshot(tvRB);

pictureBox1.Image = bmpTemp;

pictureBox2.Image = bmpTemp;

pictureBox3.Image = bmpTemp;

//if (bmpTemp != null)

// counter++;

//if (counter %2 != 0)

//{

// bmpTemp1 = bmp1;

// ShowRBTree();

// bmp1 = getControlScreenshot(treeView1);

// pictureBox1.Image = bmpTemp1;

//}

}

catch (Exception ex)

{

MessageBox.Show("Неверно введенные данные!", "Ошибка ввода", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

private void add\_5\_node\_Click(object sender, EventArgs e)

{

Class.RBTreeNode<int, string> node;

int i, k;

Random rnd = new Random();

for (i = 1; i <= 5; i++)

{

k = rnd.Next(100);

node = new Class.RBTreeNode<int, string>(k, k.ToString());

tree.InsertNode(node);

}

bmpTemp = bmp;

ShowRBTree();

bmp = getControlScreenshot(tvRB);

pictureBox1.Image = bmpTemp;

}

private void del\_Click(object sender, EventArgs e)

{

Class.RBTreeNode<int, string> node;

int i, k;

try { i = Convert.ToInt32(txtKey.Text); }

catch { i = -1; }

node = tree.Find(i, out k);

if (k == 0)

{

tree.DeleteNode(node);

bmpTemp = bmp;

ShowRBTree();

bmp = getControlScreenshot(tvRB);

pictureBox1.Image = bmpTemp;

MessageBox.Show("Элемент исключем " + node.Key + " удален!", "Удаление элемента", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);

}

else MessageBox.Show("Элемент ненайден!", "Удаление элемента", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);

}

private void Clr\_Click(object sender, EventArgs e)

{

txtKey.Text = null;

textBox3.Text = null;

textBoxKey1.Text = null;

textBoxData1.Text = null;

textBoxNumber.Text = null;

tree.Root = null;

ShowRBTree();

pictureBox1.Image = null;

pictureBox2.Image = null;

pictureBox3.Image = null;

bmp = null;

bmpTemp = null;

bmpTemp1 = null;

bmpTemp2 = null;

lbl\_pictureLabel.Visible = false;

}

private void find\_Click(object sender, EventArgs e)

{

Class.RBTreeNode<int, string> node;

int i, k;

try

{

i = Convert.ToInt32(textBox3.Text);

node = tree.Find(i, out k);

if (k == 0)

{

MessageBox.Show("Элемент с ключем " + node.Key + " найден!\nDATA = " + node.Data.ToString(), "Поиск элемента", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);

}

else

{

MessageBox.Show("Элемент с ключем " + i + " не найден!", "Поиск элемента", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);

}

}

catch

{

MessageBox.Show("Элемент ненайден!", "Поиск элемента", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);

}

}

private void radioButton2\_CheckedChanged(object sender, EventArgs e)

{

groupBoxOne.Enabled = false;

groupBoxMany.Enabled = true;

groupBoxHelp.Enabled = true;

groupBox2.Enabled = true;

groupBox3.Enabled = true;

tree.Root = null;

ShowRBTree();

pictureBox1.Image = null;

pictureBox2.Image = null;

pictureBox3.Image = null;

bmp = null;

bmpTemp = null;

lbl\_pictureLabel.Visible = false;

}

private void radioButton1\_CheckedChanged(object sender, EventArgs e)

{

this.Height = 1000;

this.Width = 1288;

groupBoxOne.Enabled = true;

groupBoxMany.Enabled = false;

groupBoxHelp.Enabled = true;

groupBox2.Enabled = true;

groupBox3.Enabled = true;

tree.Root = null;

ShowRBTree();

pictureBox1.Image = null;

pictureBox2.Image = null;

pictureBox3.Image = null;

bmp = null;

bmpTemp = null;

lbl\_pictureLabel.Visible = false;

}

private void button2\_Click(object sender, EventArgs e)

{

try

{

if (rb5 == false && rb6 == false)

MessageBox.Show("Не выбранно количество деревьев для показа", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);

else

{

int emountOfVertex = Convert.ToInt32(textBoxNumber.Text);

Class.RBTreeNode<int, string> node;

int i, k;

Random rnd = new Random();

for (i = 1; i <= emountOfVertex; i++)

{

k = rnd.Next(100);

node = new Class.RBTreeNode<int, string>(k, k.ToString());

tree.InsertNode(node);

}

//bmpTemp = bmp;

//ShowRBTree();

//bmp = getControlScreenshot(tvRB);

bmpTemp2 = bmpTemp1;

bmpTemp1 = bmpTemp;

bmpTemp = bmp;

ShowRBTree();

bmp = getControlScreenshot(tvRB);

pictureBox1.Image = bmpTemp;

pictureBox3.Image = bmpTemp2;

pictureBox2.Image = bmpTemp1;

pictureBox1.Image = bmpTemp;

}

}

catch (Exception ex)

{

MessageBox.Show("Неверно введенные данные!", "Ошибка ввода", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

int counter = 0;

private void button1\_Click\_1(object sender, EventArgs e)

{

Class.RBTreeNode<int, string> node;

int key;

string data;

//for view picture

try

{

if (rb5 == false && rb6 == false)

MessageBox.Show("Не выбранно количество деревьев для показа", "Ошибка", MessageBoxButtons.OK, MessageBoxIcon.Asterisk);

else

{

key = Convert.ToInt32(textBoxKey1.Text);

data = textBoxData1.Text.ToString();

node = new Class.RBTreeNode<int, string>(key, data);

tree.InsertNode(node);

//bmpTemp = bmp;

//ShowRBTree();

//bmp = getControlScreenshot(tvRB);

//pictureBox1.Image = bmpTemp;

bmpTemp2 = bmpTemp1;

bmpTemp1 = bmpTemp;

bmpTemp = bmp;

ShowRBTree();

bmp = getControlScreenshot(tvRB);

pictureBox1.Image = bmpTemp;

pictureBox3.Image = bmpTemp2;

pictureBox2.Image = bmpTemp1;

}

}

catch (Exception ex)

{

MessageBox.Show("Неверно введенные данные!", "Ошибка ввода", MessageBoxButtons.OK, MessageBoxIcon.Error);

}

}

private void radioButton5\_CheckedChanged(object sender, EventArgs e)

{

rb5 = true;

pictureBox2.Visible = false;

pictureBox3.Visible = false;

}

private void radioButton6\_CheckedChanged(object sender, EventArgs e)

{ rb6 = true;

pictureBox2.Visible = true;

pictureBox3.Visible = true; } }}

# *ПРИЛОЖЕНИЕ Б*

Экранные формы

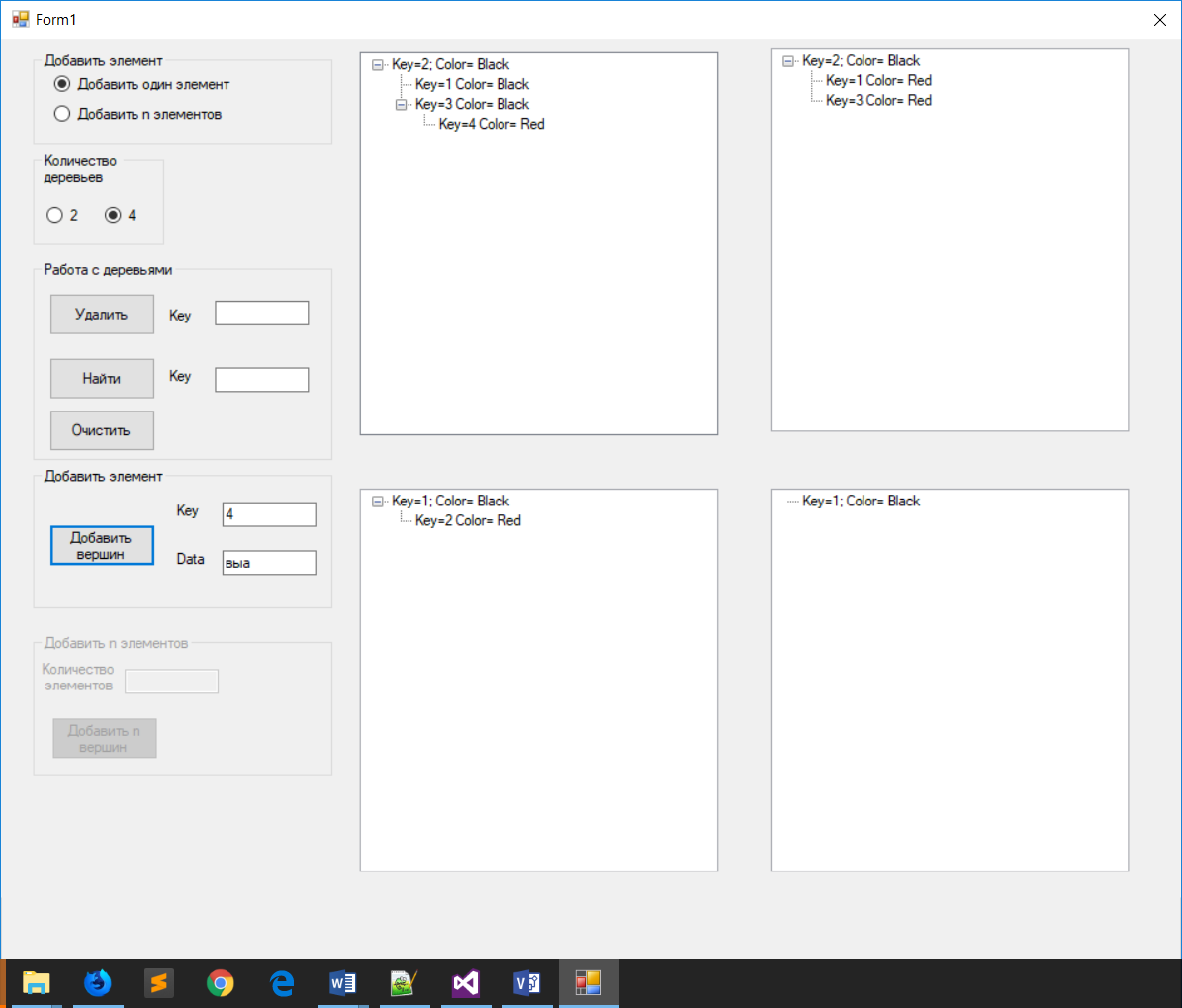


Рисунок Б.1 – Экранные формы

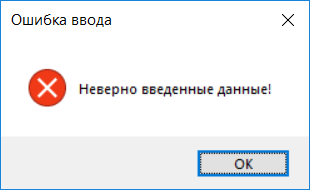
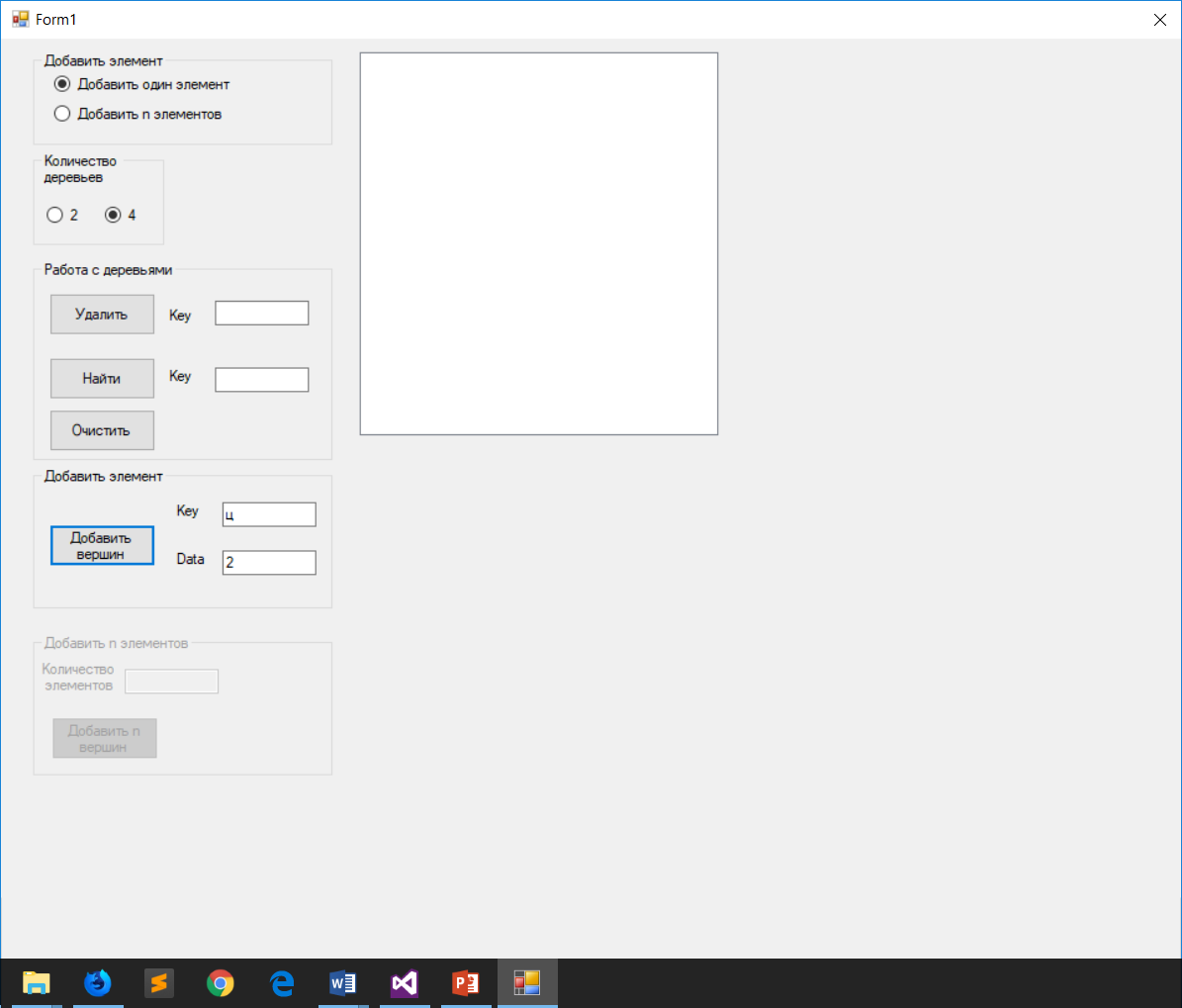


Рисунок Б.2 – Экранные формы

# *ПРИЛОЖЕНИЕ В*

Слайды презентации

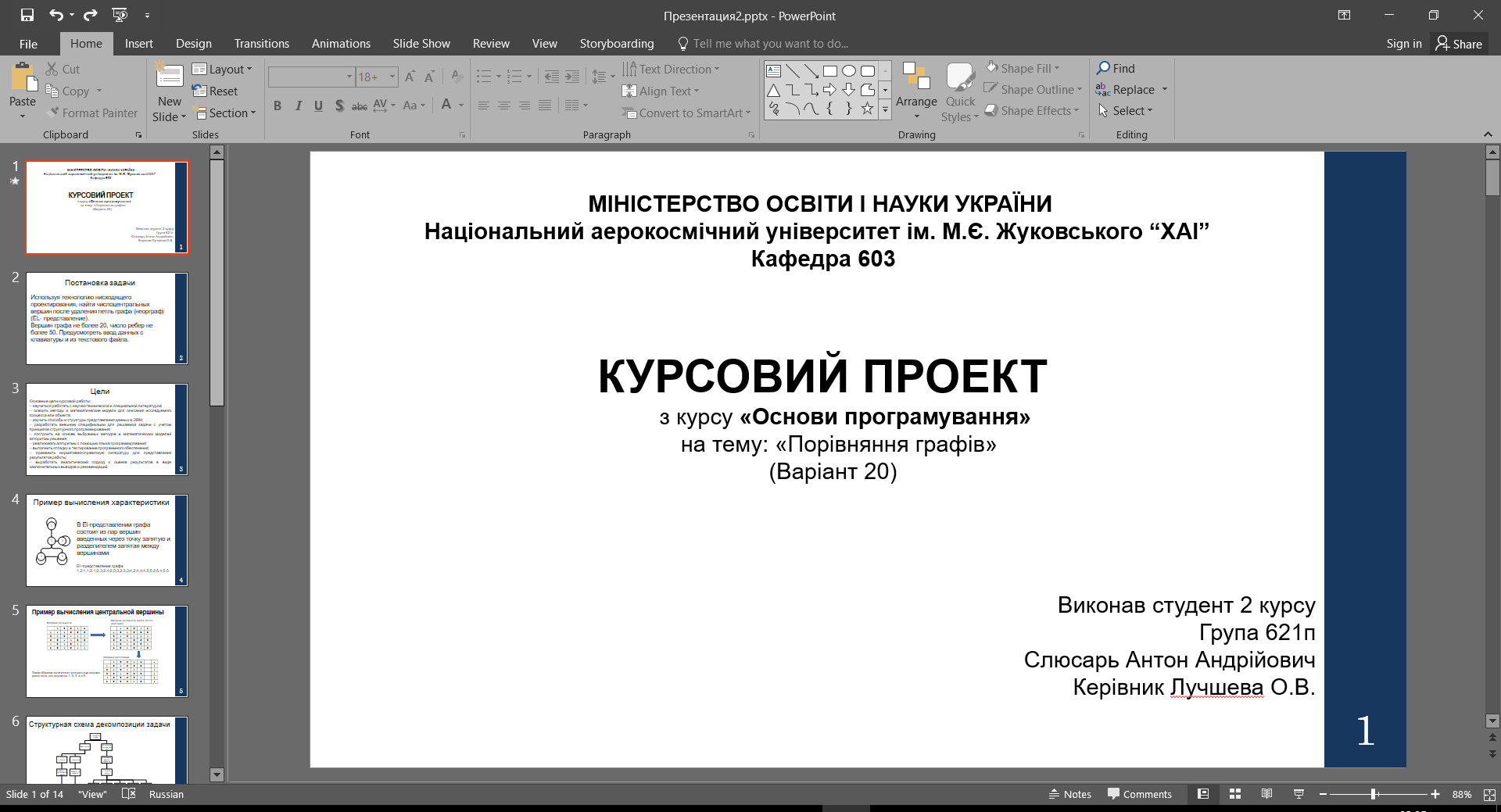


Рисунок В.1 – слайд презентации №1

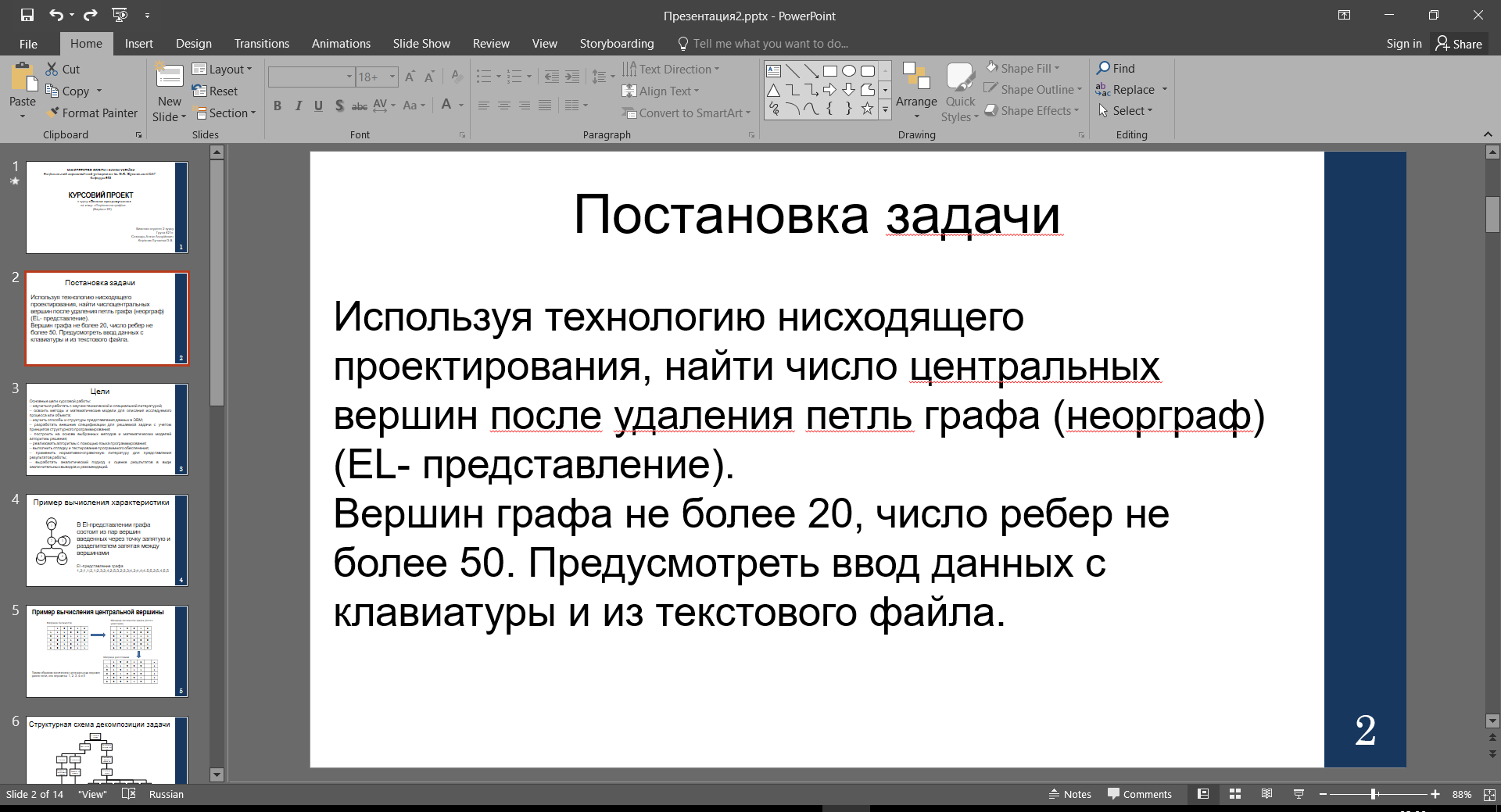


Рисунок В.2 – слайд презентации №2

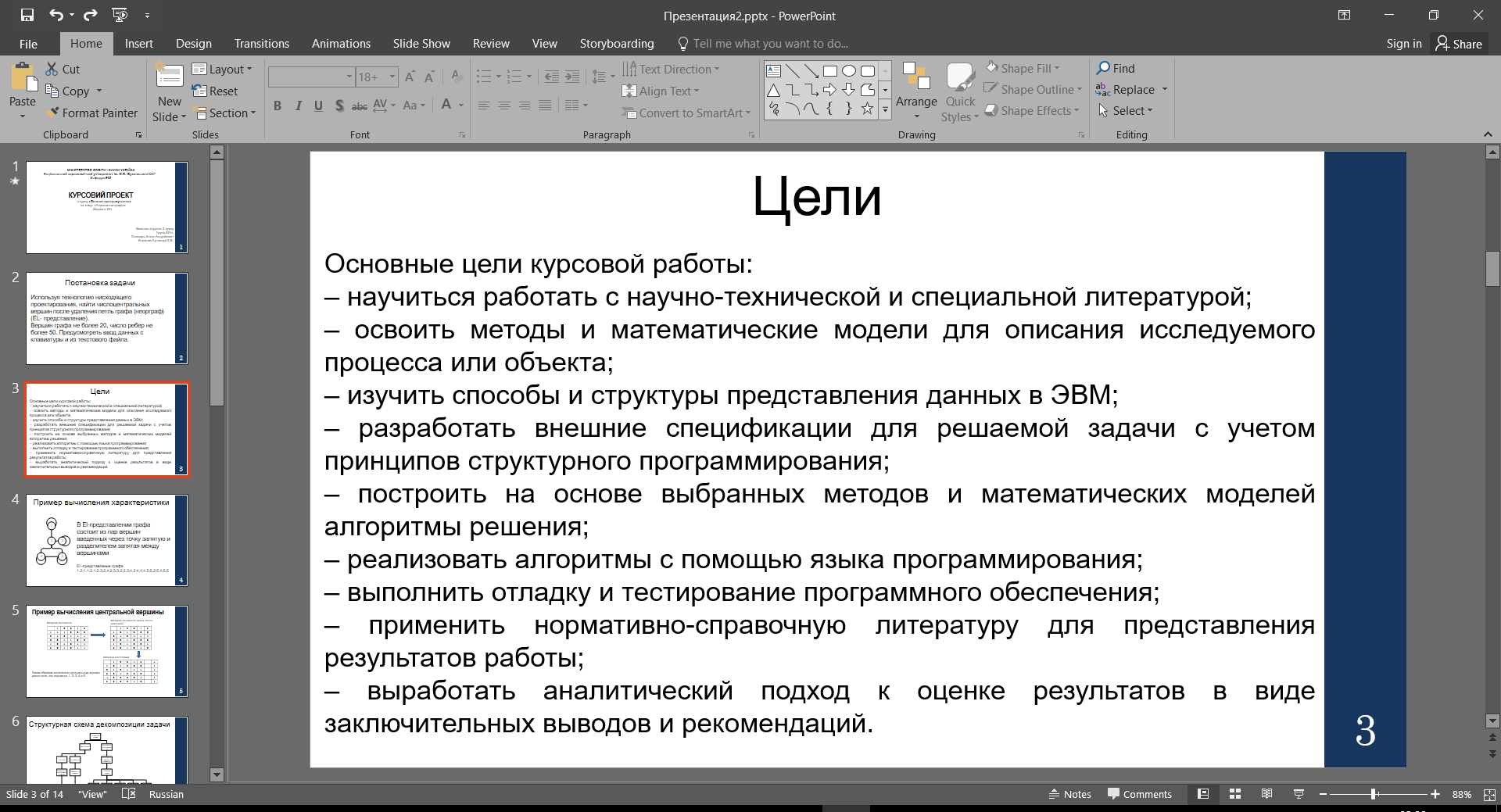


Рисунок В.3 – слайд презентации №3

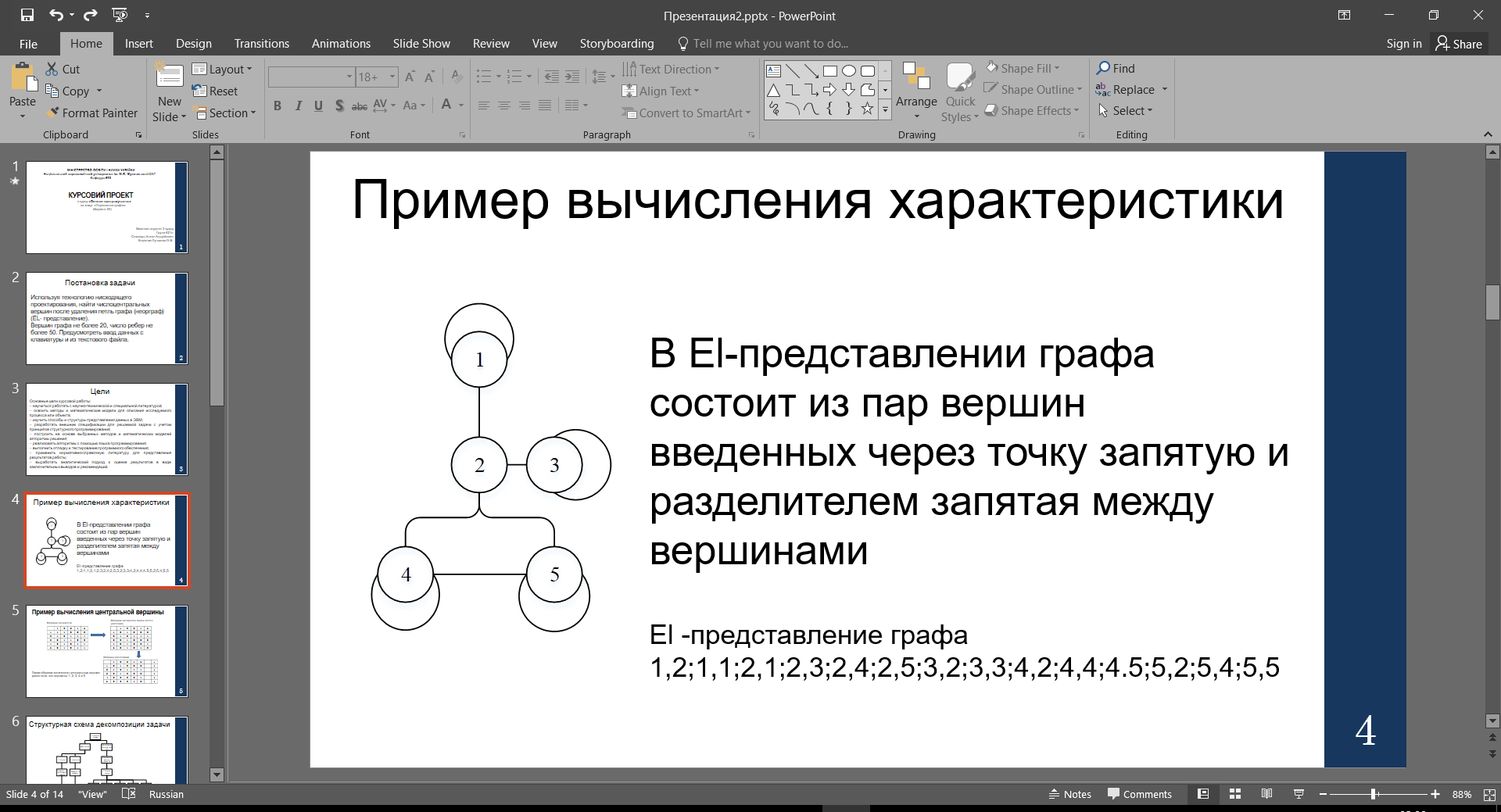


Рисунок В.4 – слайд презентации №4

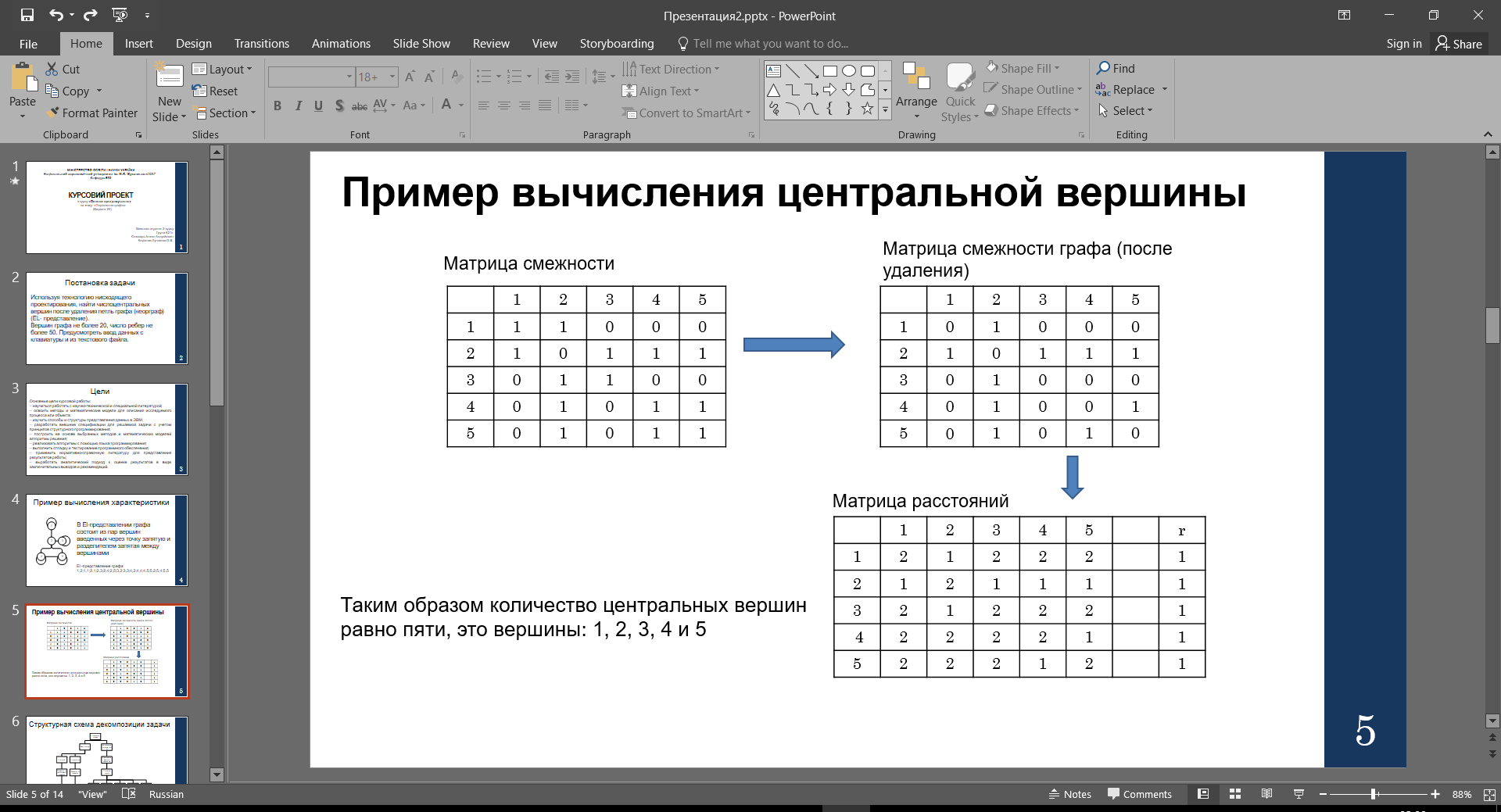


Рисунок В.5 – слайд презентации №5

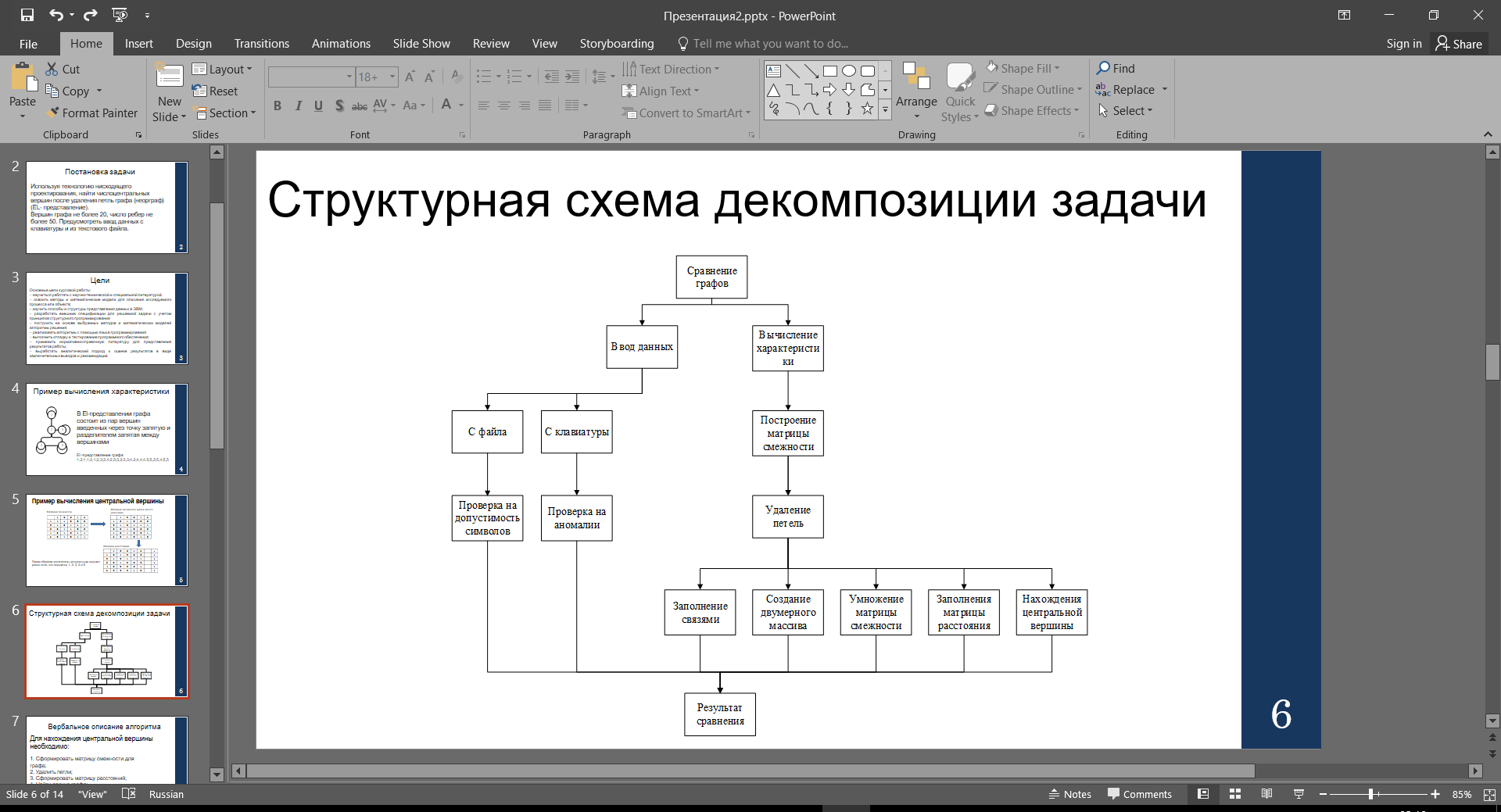


Рисунок В.6 – слайд презентации №6

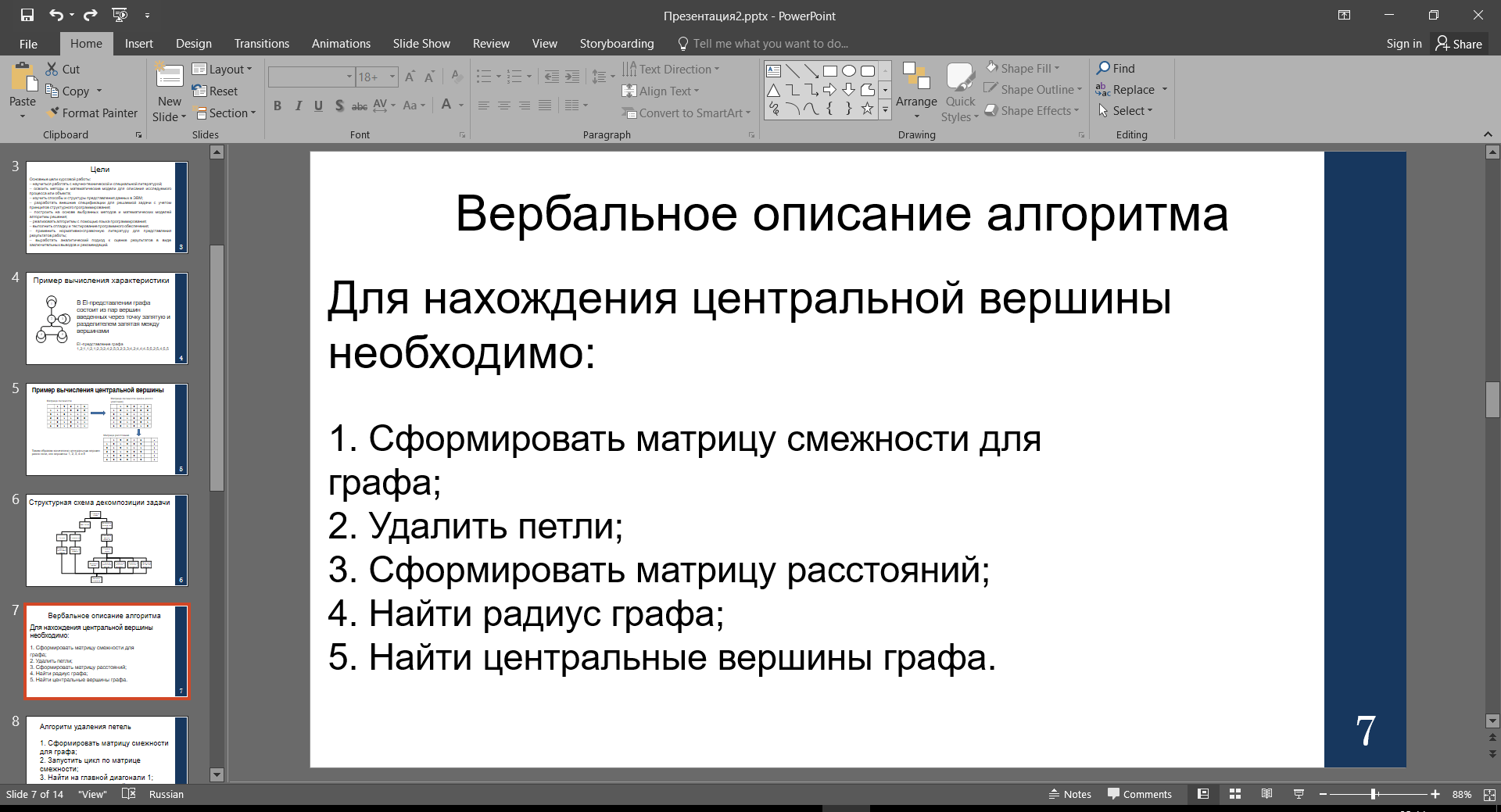


Рисунок В.7 – слайд презентации №7

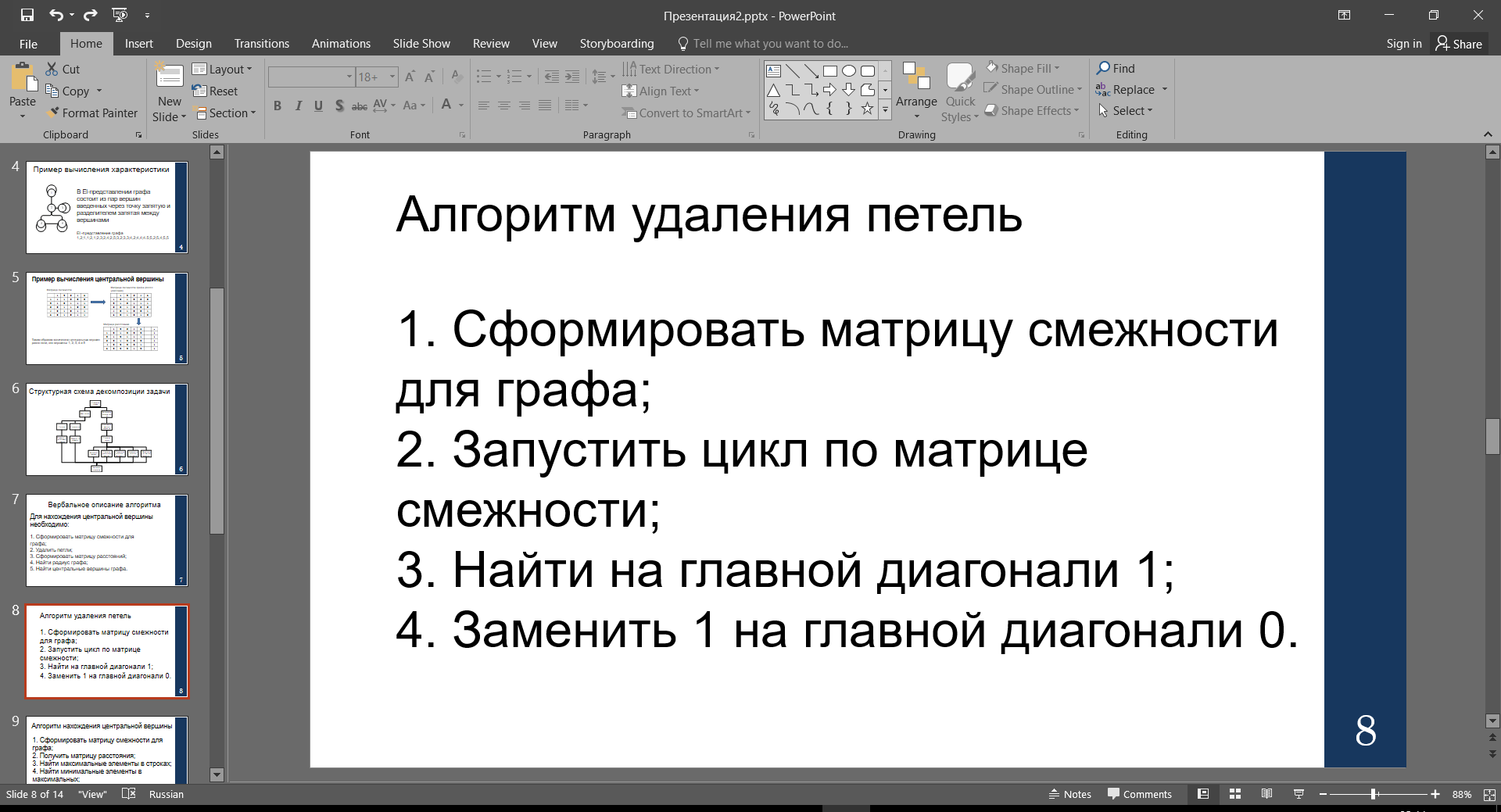


Рис В.8 – слайд презентации №8

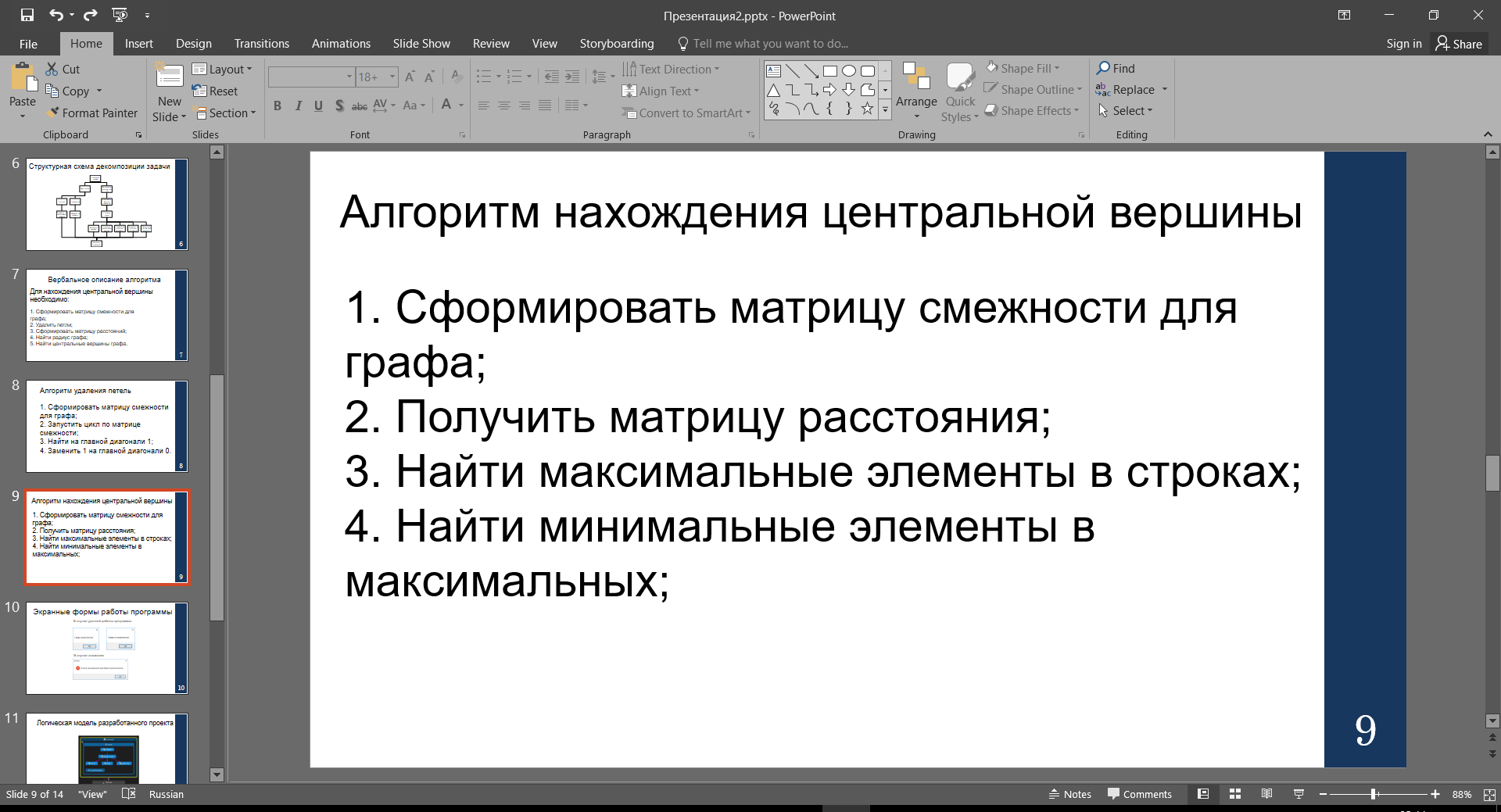


Рис В.9 – слайд презентации №9

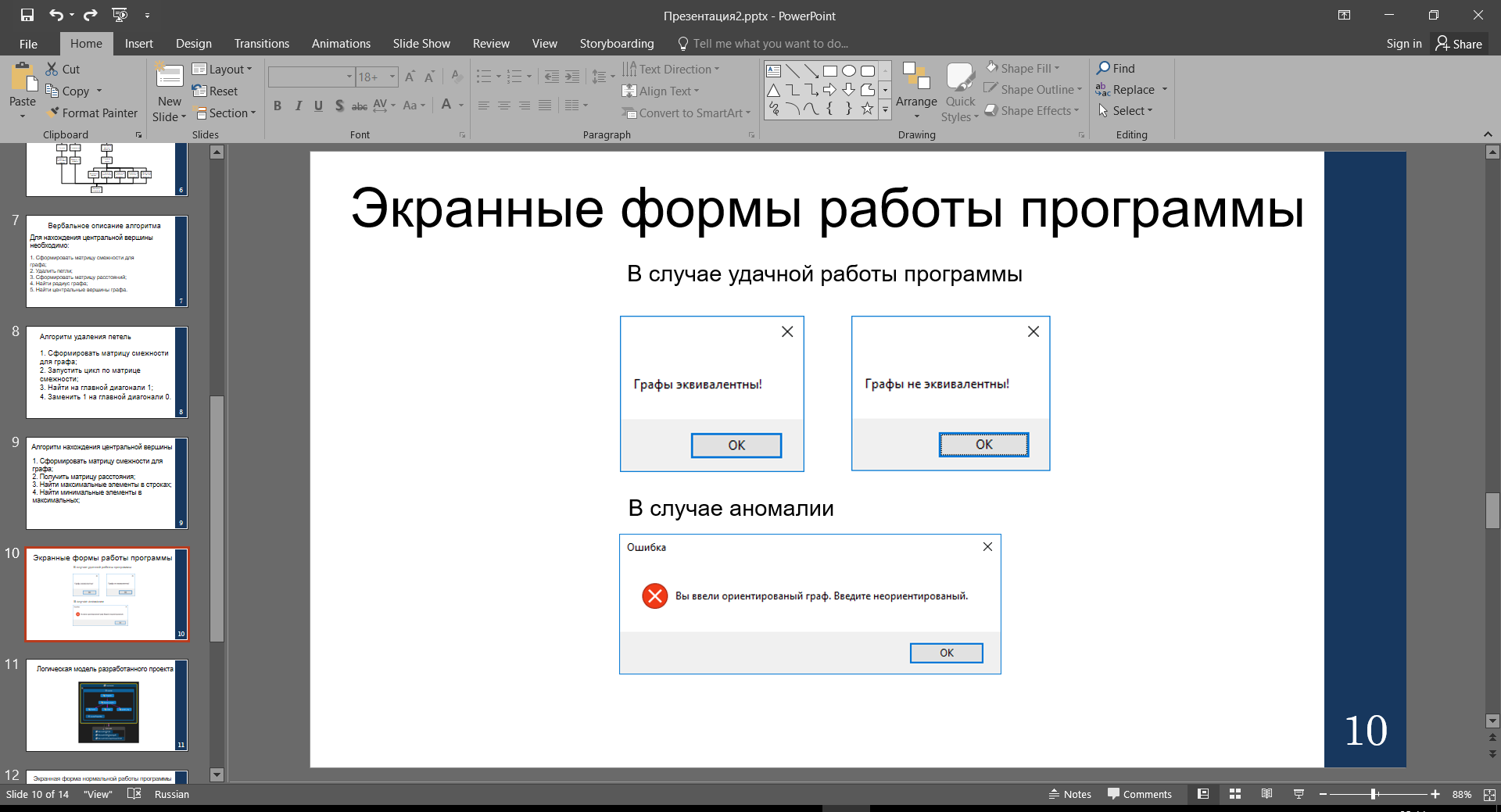


Рисунок В.10 – слайд презентации №10

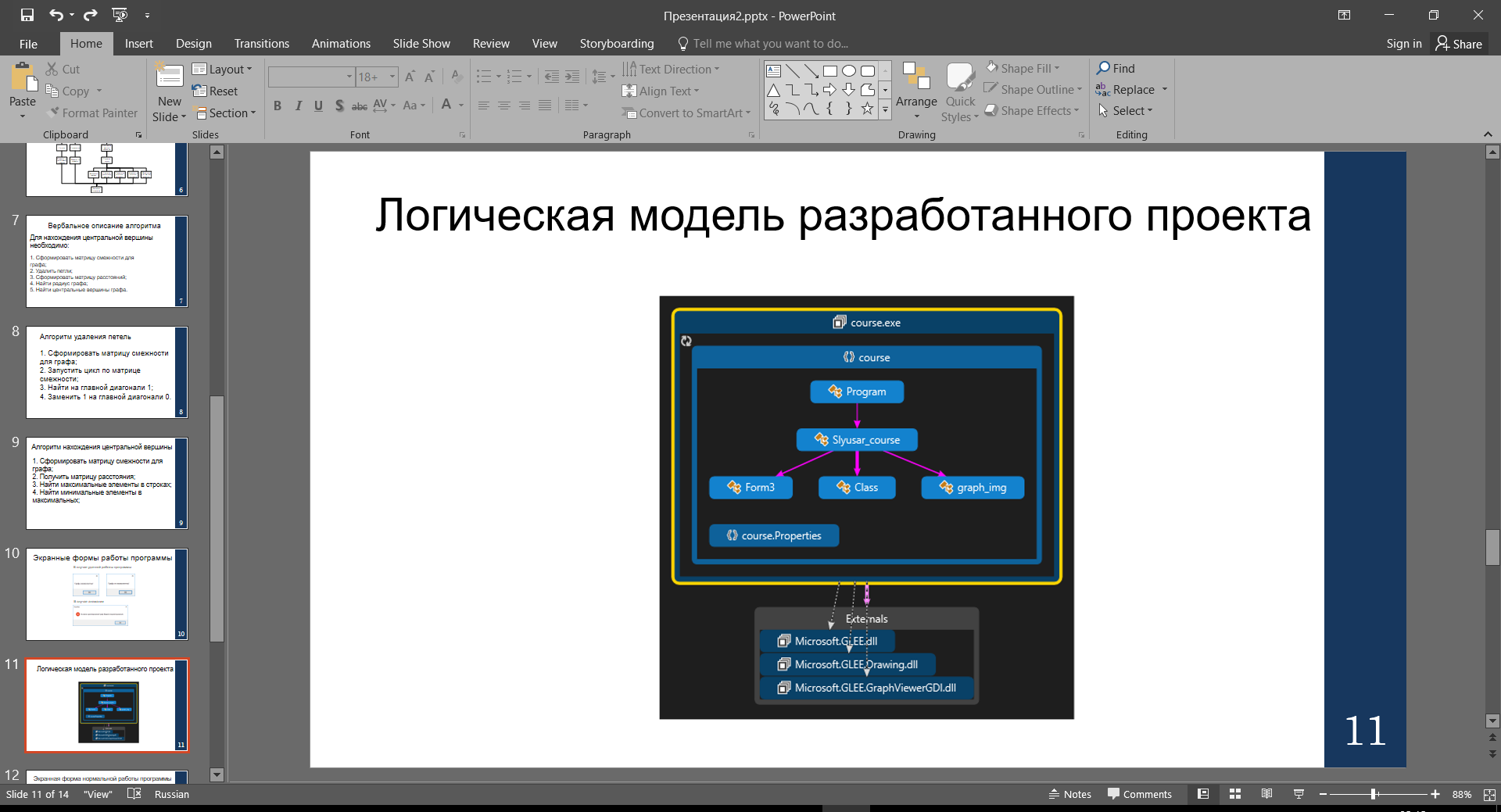


Рисунок В.11 – слайд презентации №11

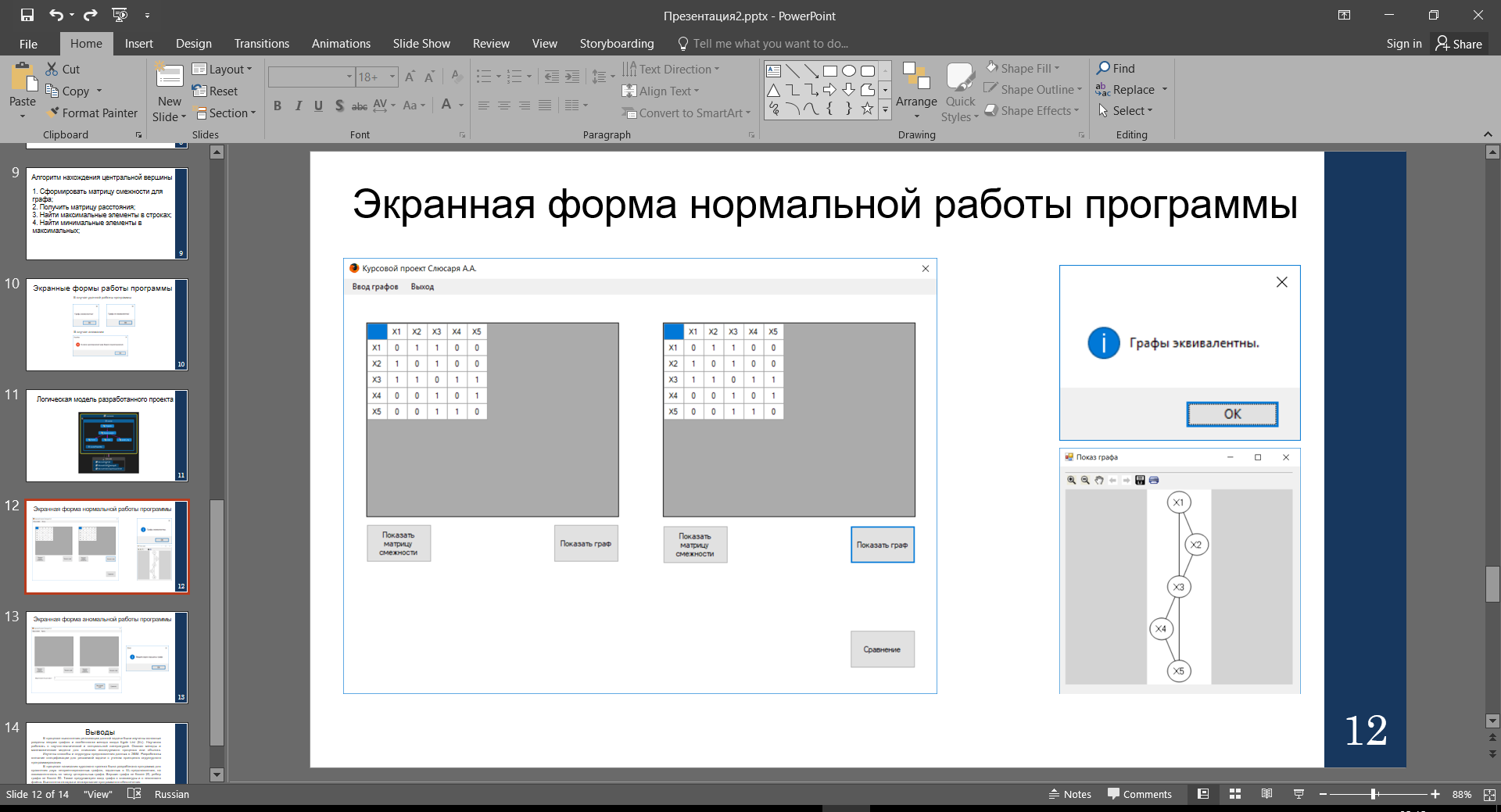


Рисунок В.12 – слайд презентации №12

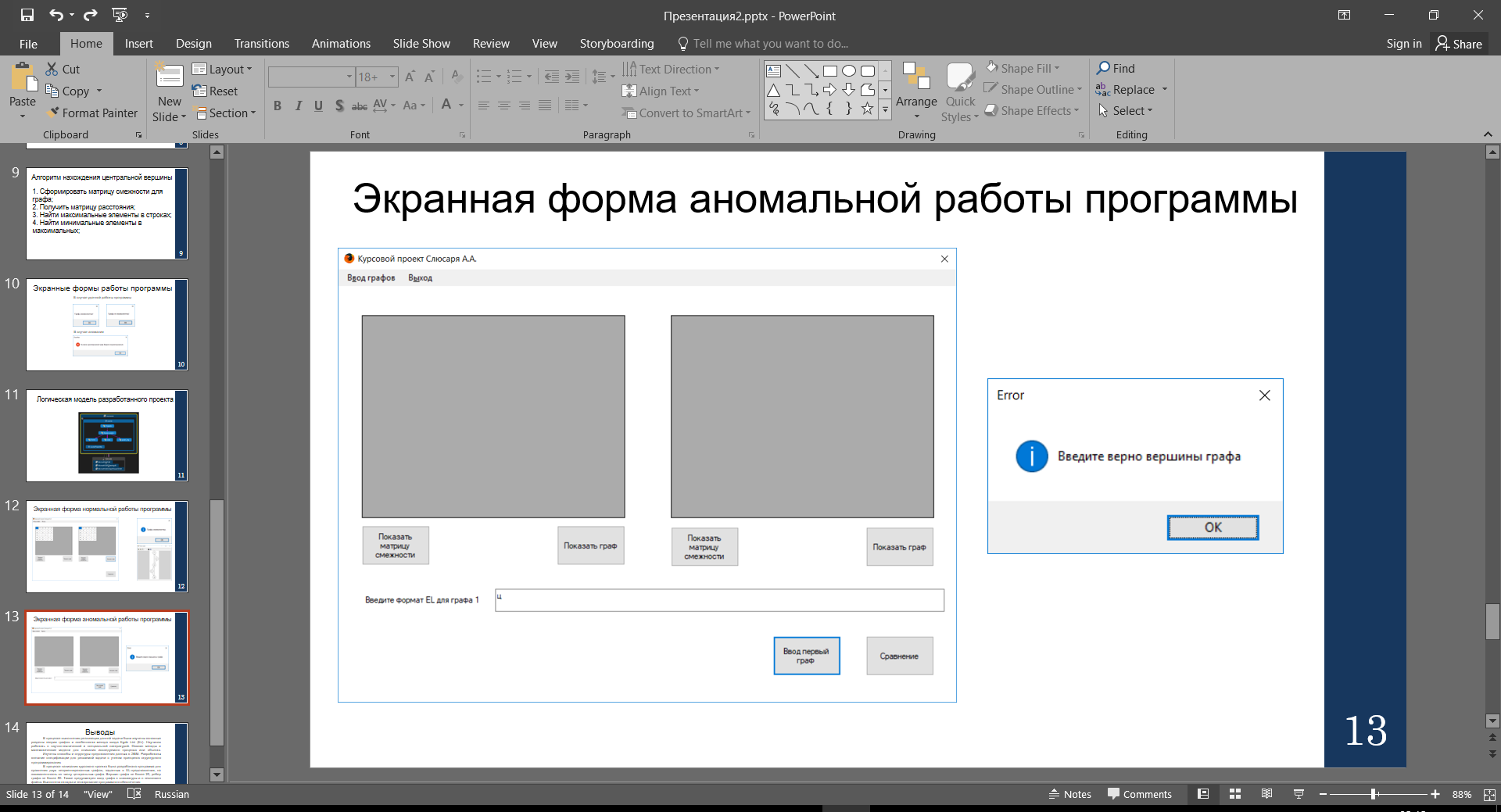


Рисунок В.13 – слайд презентации №13

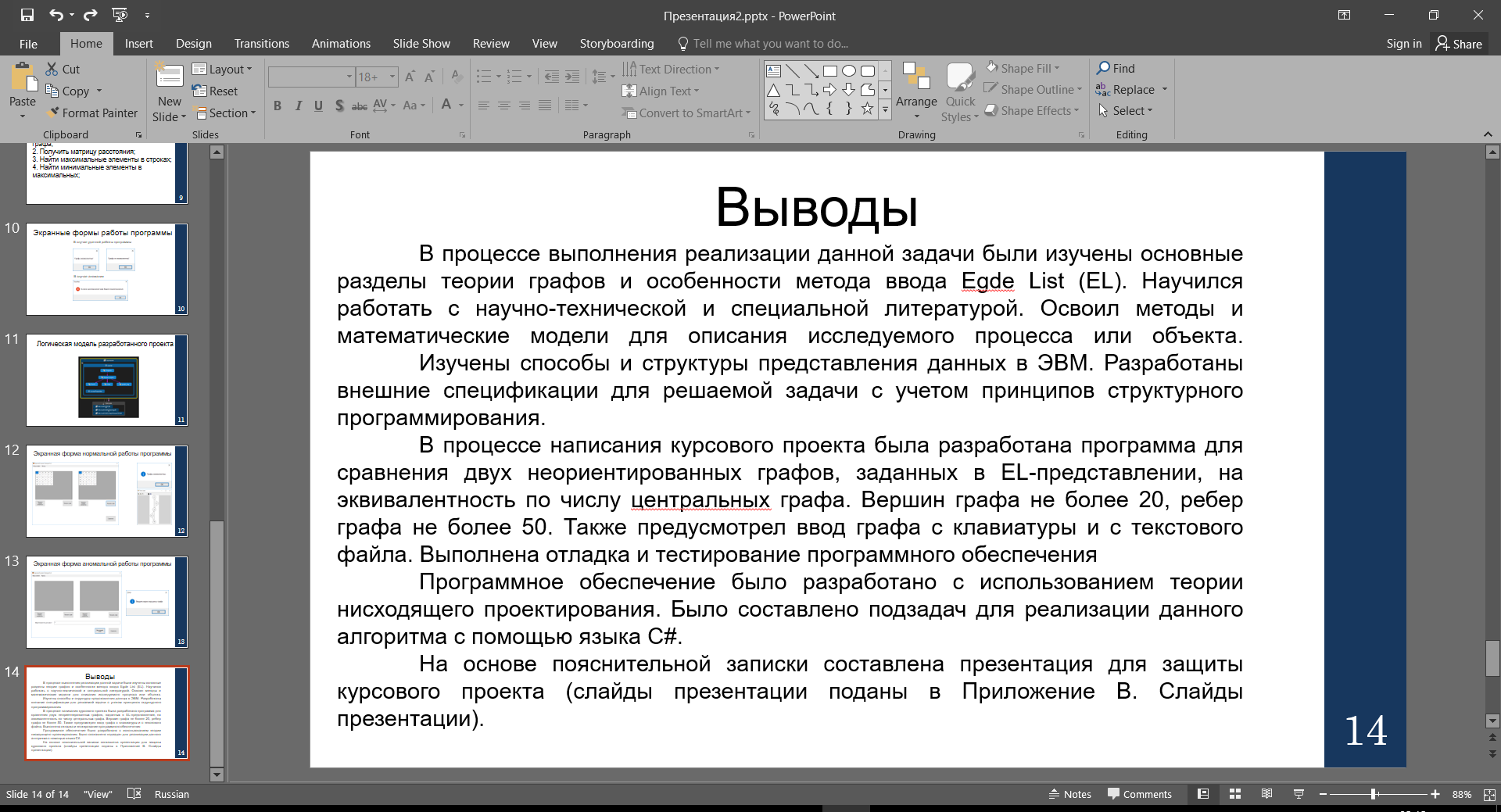


Рисунок В.14 – слайд презентации №14