Кафедра Вычислительной техники

|  |  |
| --- | --- |
| Факультет: АВТФ  Группа: АВТ-710  Студент: Покалюк Е.А. | Преподаватель:  *Ассистент каф. ВТ*  Михайленко Д.А. |

Расчетно-графическая работа

по дисциплине “Алгоритмы и структуры данных”



Оглавление

[**1.** **Общее задание** 3](#_Toc37570484)

[**2.** **Задача 1** 7](#_Toc37570485)

[**3.** **Задача 2** 7](#_Toc37570486)

[**4.** **Диаграмма взаимосвязи объектов, реализующих АТД «Простой граф», вспомогательных объектов, АТД задачи** 9](#_Toc37570487)

[**5.** **Формат АТД «Простой граф»** 9](#_Toc37570488)

[**6.** **Клиентское определение класса «Простой граф»** 15](#_Toc37570489)

[**7.** **Формат АТД «Дескриптор вершины»** 15](#_Toc37570490)

[**8.** **Клиентское определение класса «Дескриптор вершины»** 17](#_Toc37570491)

[**9.** **Формат АТД «Дескриптор ребра»** 18](#_Toc37570492)

[**10.** **Клиентское определение класса «Дескриптор ребра»** 21](#_Toc37570493)

[**11.** **Формат АТД «Итератор вершин»** 21](#_Toc37570494)

[**12.** **Клиентское определение класса «Итератор вершин»** 23](#_Toc37570495)

[**13.** **Формат АТД «Итератор ребер»** 23](#_Toc37570496)

[**14.** **Клиентское определение класса «Итератор ребер»** 25](#_Toc37570497)

[**15.** **Формат АТД «Итератор исходящих ребер»** 25](#_Toc37570498)

[**16.** **Клиентское определение класса «Итератор исходящих ребер»** 27](#_Toc37570499)

[**17.** **Формат АТД «Задача 1»** 27](#_Toc37570500)

[**18.** **Клиентское определение класса «Задача 1»** 29](#_Toc37570501)

[**19.** **Формат АТД «Задача 2»** 29](#_Toc37570502)

[**20.** **Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 1** 29](#_Toc37570503)

[**21.** **Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 2** 29](#_Toc37570504)

[**22.** **Заключение** 29](#_Toc37570505)

[**23.** **Список использованных источников** 30](#_Toc37570506)

[**24.** **Приложения** 30](#_Toc37570507)

1. **Общее задание**

Спроектировать и реализовать шаблонный класс для коллекции «Простой граф» и использовать коллекцию для решения задач для неориентированных, ориентированных и взвешенных графов.

***Разработать АТД «Простой граф».***

Интерфейс АТД «Простой граф» включает операции:

**Конструктор ( )** по умолчанию: создает пустой L - граф с нулевым числом вершин и ребер,

**Конструктор (V, D, F)** создает граф с **V** вершинами, без ребер, типа **D** (ориентированный / неориентированный), формы представления **F** (L- граф/M-граф),

**Конструктор (V, E, D, F)** создает граф с **V** вершинами, с **E** случайными ребрами, типа

**D** (ориентированный / неориентированный), формы представления **F** (L- граф/M-граф),

**Конструктор (G) -** конструктор копирования создает объект – копию графа **G,**

**Деструктор ( )** уничтожает внутренние структуры графа,

**V ( )** - возвращает число вершин в графе,

**E ( )** - возвращает число ребер в графе**,**

**Directed** ( ) - возвращает тип графа (ориентированный / неориентированный)

**Dense ( )** - возвращает форму представления графа (L- граф / M- граф),

**K ( ) -** возвращает коэффициент насыщенности графа,

**ToListGraph ( )** преобразует граф к L- графу,

**ToMatrixGraph ( )** преобразует граф к M- графу,

**InsertV ( )** добавляет вершину к графу и возвращает адрес дескриптора вновь созданной вершины,

**DeleteV (*v*) -** удаляет вершину из графа, заданную адресом дескриптора ***v***,

**InsertE(*v1, v2*) -** добавляет ребро (***v1, v2***)к графу, соединяющую вершины, заданные адресами дескрипторов ***v1*** и ***v2,*** и возвращает адрес дескриптора вновь созданного ребра - ***e***,

**DeleteE (*v1, v2*)** удаляет ребро, соединяющее вершины, заданные адресами дескрипторов ***v1*** и ***v2,***

**GetEdge (*v1, v2*)** возвращает адрес дескриптора ребра соединяющего вершины, заданные дескрипторами ***v1*** и ***v2***,

***Разработать ассоциированные с графом типы:***

***АТД «Дескриптор вершины графа»***

**Дескриптор вершины содержит поля:**

***name –*** имя вершины,

***data –*** данные, связанные с вершиной,

***index –*** индекс вершины в структуре графа или -1,

**Интерфейс АТД «Дескриптор вершины графа» включает операции:**

**Конструктор** ():поле ***name*** не определено, поле ***data*** не определено,

**Конструктор** (***name, data***): ***name*** - имя вершины, ***data*** - данные, связанные с вершиной,

**GetName ( ) -** возвращает имя вершины,

**GetData ( ) -** возвращает данные, связанные с вершиной,

**SetName (name ) –** задает имя вершины,

**SetData (*data*) –** записывает данные ***data*** в дескриптор вершины.

***АТД «Дескриптор ребра графа»***

**Дескриптор ребра содержит поля:**

***v1*** -дескриптор вершины, из которой исходит ребро,

***v2*** -дескриптор вершины, в которую входит ребро,

***w*** -вес ребра,

***data*** - данные, связанные с ребром,

**Интерфейс АТД «Дескриптор ребра графа» включает операции:**

**Конструктор** (***v1, v2***): ***v1*** -дескриптор вершины, из которой исходит ребро, ***v2*** -дескриптор вершины, в которую входит ребро,

**Конструктор** (***v1, v2, w***): ***v1*** -дескриптор вершины, из которой исходит ребро, ***v2*** -дескриптор вершины, в которую входит ребро, ***w*** -вес ребра,

**Конструктор** (***v1, v2, w, data***): ***v1*** -дескриптор вершины, из которой исходит ребро, ***v2*** -дескриптор вершины, в которую входит ребро, ***w*** -вес ребра, ***data*** - данные, связанные с ребром

**v1( ) -** возвращает дескриптор вершины, из которой исходит ребро,

**v2( ) -** возвращает дескриптор вершины, в которую входит ребро,

**GetW ( ) -** возвращает вес ребра,

**SetW (*w*) -** изменение веса ребра,

**GetData** ( ) - возвращает данные, связанные с ребром,

**SetData** (***data***) - изменение данных, связанных с ребром.

***АТД «Итератор вершин графа»***

**Интерфейс АТД «Итератор вершин графа» включает операции:**

**Конструктор** () - создает итератор вершин графа,

**beg ( )** - возвращает итератор, установленный на первую вершину графа,

**end (** ) - возвращает итератор, соответствующий окончанию переходов итератора,

**operator ++** - переход к следующей вершине графа,

**operator \*** - возвращает дескриптор вершины графа, на которую указывает итератор.

***АТД «Итератор ребер графа»***

**Интерфейс АТД «Итератор ребер графа» включает операции:**

**Конструктор** () - создает итератор ребер графа,

**beg ( )** - возвращает итератор, установленный на первое ребро графа,

**end (** ) - возвращает итератор, соответствующий окончанию переходов итератора,

**operator ++** - переход к следующему ребру графа,

**operator \*** - возвращает дескриптор ребра графа, на которое указывает итератор.

***АТД «Итератор исходящих ребер вершины»***

**Интерфейс АТД «Итератор исходящих ребер вершины» включает операции:**

**Конструктор** (***v***) - создает итератор исходящих ребер графа для вершины, заданной дескриптором ***v***,

**beg ( )** - возвращает итератор, установленный на первое исходящее ребро вершины,

**end (** ) - возвращает итератор, соответствующий окончанию переходов итератора,

**operator ++** - переход к следующему исходящему ребру,

**operator \*** - возвращает дескриптор исходящего ребра вершины, на которое указывает итератор.

1. **Задача 1**

Спроектировать и реализовать шаблонный класс для ***АТД «Задача 1»*** в соответствии с вариантом и использовать для решения задачи на неориентированном или ориентированном графе.

Вариант 16: Формирование списка ребер неориентированного графа в порядке двухпроходного Эйлерова цикла.

Спроектировать и реализовать шаблонный класс для ***АТД «Задача 1»*** в соответствии с вариантом и использовать для решения задачи на неориентированном или ориентированном графе.

Интерфейс ***АТД «Задача 1»*** включает операции:

**Конструктор** (***g***) - создает объект задачи 1, ассоциированный с графом ***g***, и выполняет решение задачи для графа ***g***,

**Конструктор (*T*) -** конструктор копирования создает копию объекта – задачи ***T*,**

**Деструктор ( ) -** уничтожает внутренние структуры объекта задачи,

**Set** (***g***) – связывает объект задачи с графом ***g*** и выполняет решение задачи 1 для графа ***g***,

**Restart** ( ) – повторно выполняет решение задачи 1 для графа ***g***,

**Result ( )** – возвращает результат решения задачи 1

1. **Задача 2**

Вариант 4: Определение центра взвешенного орграфа на основе алгоритма Беллмана – Форда.

Спроектировать и реализовать шаблонный класс для ***АТД «Задача 2»*** в соответствии с вариантом и использовать для решения задачи на взвешенном графе.

Интерфейс ***АТД «Задача 3»*** включает операции:

**Конструктор (g)** - создает объект задачи 3, ассоциированный с графом g, и выполняет решение задачи для графа g,

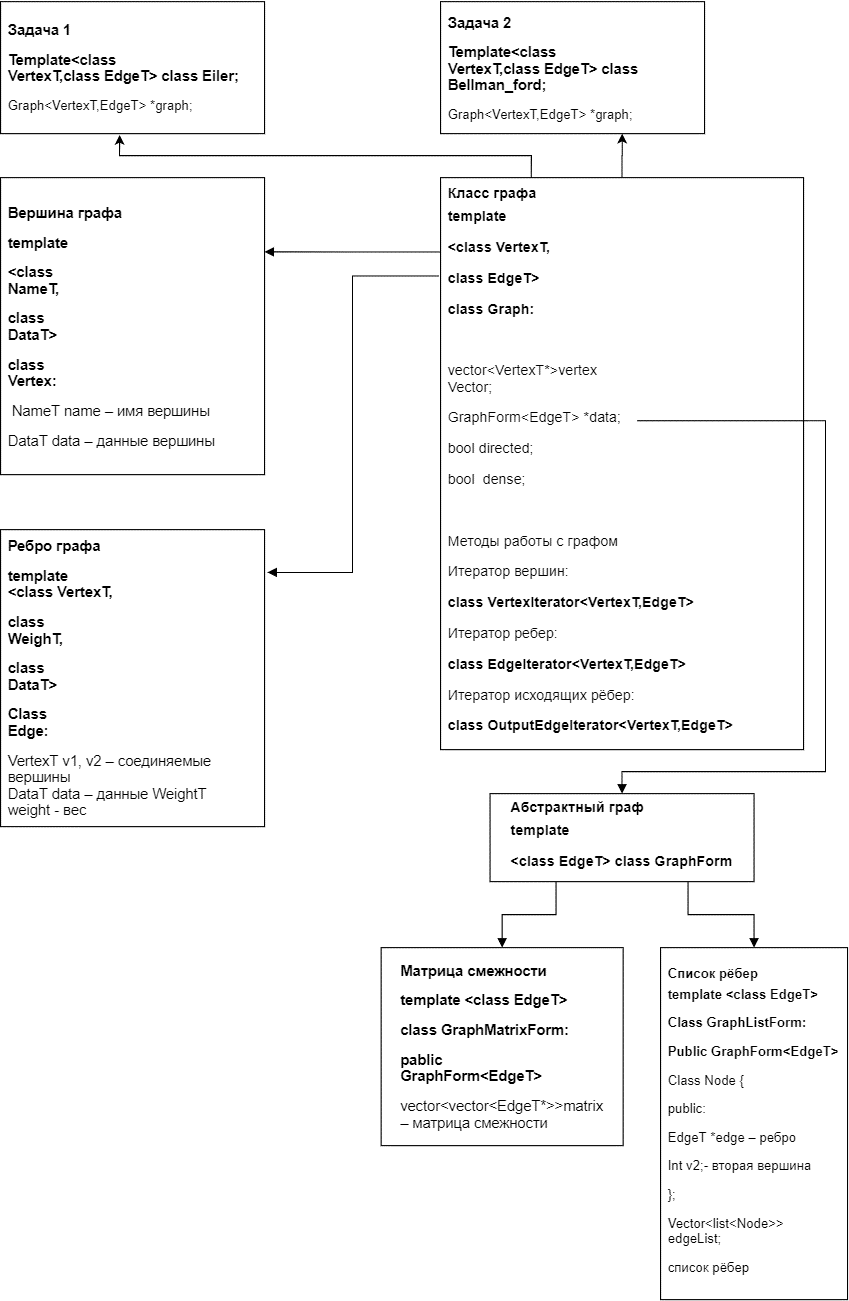
**Конструктор (*T*) -** конструктор копирования создает копию объекта – задачи ***T*,**

**Деструктор ( ) -** уничтожает внутренние структуры объекта задачи,

**Set (g)** – связывает объект задачи с графом g и выполняет решение задачи 3 для графа g,

**Restart ( )** – повторно выполняет решение задачи 3 для графа g,

**Result( )** – возвращает результат решения задачи 3

1. **Диаграмма взаимосвязи объектов, реализующих АТД «Простой граф», вспомогательных объектов, АТД задачи**
2. **Формат АТД «Простой граф»**

Простой граф представляет собой граф, который может иметь структуру, представленную либо в виде матрицы смежностей, либо в виде списка смежностей. Граф может быть ориентированным и неориентированным. Класс предоставляет доступ к хранимым в нём вершинам и рёбрам, типы которых задаются пользователем. Доступ к вершинам осуществляется по их дескрипторам. Доступ к рёбрам осуществляется со дескриптору исходящей вершины ребра и дескриптору входящей вершины ребра.

**ДАННЫЕ:**

*Параметры:*

Коллекция вершин;

Коллекция рёбер;

Тип графа **dense;**

Ориентированность графа **directed**.

*Структура данных:*

Матрица/Массив списков.

**ОПЕРАЦИИ:**

*Конструктор*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: создание пустого неориентированного L-графа.

**Выход**: нет.

**Постусловия**: создан пустой граф.

*Конструктор*

**Вход**: количество вершин V, ориентированность D, форма F.

**Предусловия**: нет.

**Процесс**: создание графа типа F, ориентированности D, содержащего V вершин.

**Выход**: нет.

**Постусловия**: создан граф с V вершинами.

*Конструктор*

**Вход**: количество вершин V, количество рёбер Е,ориентированность D, форма F.

**Предусловия**: нет.

**Процесс**: создание графа типа F, ориентированности D, содержащего V вершин и Е случайно сгенерированных рёбер.

**Выход**: нет.

**Постусловия**: создана коллекция с V вершинами и Е случайными рёбрами.

*Конструктор копирования*

**Вход**: копируемый граф G.

**Предусловия**: нет.

**Процесс**: копирование данных графа G.

**Выход**: нет.

**Постусловия**: коллекция скопирована.

*Деструктор*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: удаление коллекции.

**Выход**: нет.

**Постусловия**: коллекция удалена.

*Получение числа вершин в графе*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: получение размера коллекции вершин графа.

**Выход**: число вершин в графе.

**Постусловия**: нет.

*Получение числа рёбер в графе*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: получение размера коллекции рёбер графа.

**Выход**: число рёбер в графе.

**Постусловия**: нет.

*Получение коэффициента заполенности графа*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: подсчёт коэффициента корреляции.

**Выход**: коэффициент корреляции графа.

**Постусловия**: нет.

*Получение направленности графа*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка параметра **directed**.

**Выход**: true, если граф ориентированный, false, если неориентированный.

**Постусловия**: нет.

*Получение формы представления графа*

**Вход***:* нет.

**Предусловия**: нет.

**Процесс**: проверка параметра **dense.**

**Выход**: true, если граф М типа, false, если L типа.

**Постусловия**: нет.

*Преобразование графа к М-типу*

**Вход**: нет

**Предусловия**: нет

**Процесс**: преобразование внутренней структуры графа к М-типу

**Выход**: нет

**Постусловия**: граф принял M-форму.

*Преобразование графа к L-типу*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: преобразование внутренней структуры графа к L-типу.

**Выход**: нет.

**Постусловия**: граф принял L-форму.

*Вставка вершины в граф*

**Вход***:* нет.

**Предусловия***:* нет.

**Процесс***:* создание новой вершины и её вставка в конец коллекции вершин.

**Выход***:* дескриптор созданной вершины.

**Постусловия***:* в граф добавлена новая вершина.

*Удаление вершины из графа*

**Вход**: дескриптор удаляемой вершины V.

**Предусловия**: вершина V содержится в графе.

**Процесс**: удаление вершины с дескриптором V из графа.

**Выход**: true, при выполнении предусловия, иначе false.

**Постусловия**: из графа удалена вершина V.

*Получение вершины из графа*

**Вход**: имя вершины **name**.

**Предусловия**: проверка на наличие вершины с именем **name** в графе.

**Процесс**: получение дескриптора вершины V с именем **name**.

**Выход**: дескриптор вершины при выполнении предусловия, иначе генерация исключения.

**Постусловия**: нет.

*Вставка ребра в граф*

**Вход**: дескрипторы исходящей вершины V1 и входящей вершины V2, которые будут соденинены ребром.

**Предусловия**: вершины с дескрипторами V1 и V2 содержатся в графе и не соединены ребром, идущим из вершины с дескриптором V1 в вершину с дескриптором V2.

**Процесс**: создание нового ребра, исходящего из вершины с дескриптором V1 в веришну с дескрипотором V2 и её вставка в коллекцию рёбер.

**Выход**: дескриптор созданного ребра при выполнении предусловия, иначе генерация исключения.

**Постусловия**: ребро добавлено в граф.

*Удаление ребра из графа*

**Вход**: дескрипторы исходящей вершины V1 и входящей вершины V2, соединённых удаляемым ребром.

**Предусловия**: вершины с дескрипторами V1 и V2 содержатся в графе и соединены ребром, исходящим из вершины с дескриптором V1 и входящим в вершину с дескриптором V2.

**Процесс**: удаление ребра, соединяющего вершины с дескрипторами V1 и V2.

**Выход**: true, при выполнении предусловия, иначе false

**Постусловия**: из графа удалено ребро.

*Проверка на существование ребра в графе*

**Вход**: дескрипторы исходящей вершины V1 и входящей вершины V2, идентифицирующих ребро.

**Предусловия**: вершины с дескрипторами V1 и V2 содержатся в графе.

**Процесс**: проверка на существование ребра между вершинами с дескрипоторами V1 и V2.

**Выход**: true, если вершины с дескрипторами V1 и V2 соединены ребром, иначе false.

**Постусловия**: нет.

*Получение ребра из графа*

**Вход**: дескрипторы исходящей вершины V1 и входящей вершины V2, идентифицирующих ребро.

**Предусловия**: вершины с дескрипторами V1 и V2 содержатся в графе и соединены ребром, исходящим из вершины с дескриптором V1 и входящим в вершину с дескриптором V2.

**Процесс**: получение ребра между вершинами V1 и V2

**Выход**: идентификатор ребра, если выполнено предусловие, иначе генерация исключения.

**Постусловия**: нет.

1. **Клиентское определение класса «Простой граф»**

template <class VertexT, class EdgeT>

class Graph {

Graph(); //Конструктор пустого L-графа

Graph(int vertexCount, bool directed, bool dense); //Конструктор с //заданным числом вершин, заданной ориентированности и заданного типа

Graph(int vertexCount, int edgeCount, bool directed, bool dense); //Конструктор с заданным числом вершин, случайных рёбер, заданной

//ориентированности и типа

Graph(const Graph<VertexT, EdgeT> &G); //Конструктор копирования

~Graph(); //Деструктор

int getVertexCount(); // Число вершин

int getEdgeCount(); // Число рёбер

double getRatio(); // Заполненность графа

bool isDirected(); // Направленность графа

bool isDense(); // Тип графа

void toDenseGraph(); // Преобразование к M-типу

void toSparseGraph(); // Преобразование к L-типу

VertexT\* InsertV (); // Вставка новой вершины

bool deleteVertex(VertexT \*v); // Удаление вершины

EdgeT\* InsertE (VertexT \*v1, VertexT \*v2); // Вставка нового ребра

bool deleteEdge(VertexT \*v1, VertexT \*v2) // Удаление ребра

bool hasEdge(VertexT \*v1, VertexT \*v2); // Проверка на

// Существованиеие ребра

EdgeT\* getEdge(VertexT \*v1, VertexT \*v2); // Получение ребра

};

1. **Формат АТД «Дескриптор вершины»**

Разработанный абстрактный тип данных представляет собой вершину, предназначенную для вставки в граф. Вершина имеет поле имени вершины и поле данных вершины, типы которых определяются параметрами шаблона.

Получение и установка параметров вершины осуществляется с помощью её методов.

**ДАННЫЕ:**

*Параметры*

Имя вершины **name**.

Данные вершины **data**.

**ОПЕРАЦИИ:**

*Конструктор*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: создание пустой вершины с неопределёнными именем и данными.

**Выход**: нет.

**Постусловия**: создана пустая вершина.

*Конструктор*

**Вход**: имя вершины **name** и данные вершины **data**.

**Предусловия**: нет.

**Процесс**: создание вершины с именем **name** и данными **data**.

**Выход**: нет.

**Постусловия**: создана вершина с заданными именем и данными.

*Установка имени вершины*

**Вход**: имя вершины **name**.

**Предусловия**: нет.

**Процесс**: установка имени вершины в значение **name**.

**Выход**: нет.

**Постусловия**: имя вершины установлено в значение **name**.

*Установка данных вершины*

**Вход**: данные вершины **data**.

**Предусловия**: нет.

**Процесс**: установка данных вершины в значение **data**.

**Выход**: нет.

**Постусловия**: данные вершины установлены в значение **data**.

*Получение данных вершины*

**Вход**: нет.

**Предусловия**: данные были установлены.

**Процесс**: обращение к полю объекта **data**.

**Выход**: данные вершины **data** или неопределённое значение, если не выполнено предусловие.

**Постусловия**: нет

*Получение имени вершины*

**Вход**: нет.

**Предусловия**: имя было установлено.

**Процесс**: обращение к полю объекта **name**.

**Выход**: имя вершины **name**.

**Постусловия**: нет.

*Провекра установленности имени вершины*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **name**.

**Выход**: true, если имя уставнолено, иначе false.

**Постусловия**: нет.

*Провекра установленности данных вершины*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **data**.

**Выход**: true, если данные уставнолены, иначе false.

**Постусловия**: нет.

1. **Клиентское определение класса «Дескриптор вершины»**

template <class NameT, class DataT>

class Vertex {

NameT name; // Имя вершины

DataT data; // Данные вершины

bool nameSet, dataSet; // Признаки установки имени и данных

public:

Vertex(); // Конструктор пустой вершины

Vertex(NameT name, DataT data); // Конструктор с параметрами имени и данных

void setName(NameT name); // Установка имени

void setData(DataT data); // Установка данных

NameT getName(); // Получение имени

DataT getData(); // Получение данных

bool isNameSet(); // Проверка установленности имени

bool isDataSet(); // Проверка установленности данных

};

1. **Формат АТД «Дескриптор ребра»**

Абстрактный тип данных представляет собой ребро, предназначенное для соединения вершин в графе. Ребро имеет поле веса ребра, поле данных ребра, поле исходящей вершины и поле входящей вершины, типы которых определяются параметрами шаблона.

Получение и установка параметров ребра осуществляется с помощью его методов.

**ДАННЫЕ:**

*Параметры*

Вес вершины **weight**.

Данные вершины **data**.

**ОПЕРАЦИИ:**

*Конструктор*

**Вход**: дескрипторы вершин V1 и V2, которые соединяются ребром.

**Предусловия**: нет.

**Процесс**: создание ребра с неопределёнными весом и данными между V1 и V2.

**Выход**: нет.

**Постусловия**: создано ребро с неопределёнными весом и данными между V1 и V2.

*Конструктор*

**Вход**: дескрипторы вершин V1 и V2, которые соединяются ребром, вес **weight**.

**Предусловия**: нет.

**Процесс**: создание ребра с весом **weight** между V1 и V2.

**Выход**: нет.

**Постусловия**: создано ребро с весом **weight** и неопределёнными данными между V1 и V2.

*Конструктор*

**Вход**: дескрипторы вершин V1 и V2, которые соединяются ребром, вес **weight,** данные **data**.

**Предусловия**: нет.

**Процесс**: создание ребра с весом **weight** и данными **data** между V1 и V2.

**Выход**: нет.

**Постусловия**: создано ребро с весом **weight** и данными **data** между V1 и V2.

*Установка веса ребра*

**Вход**: вес ребра **weight**.

**Предусловия**: нет.

**Процесс**: установка веса ребра в значение **weight**.

**Выход**: нет.

**Постусловия**: вес ребра установлен в значение **weight**.

*Установка данных ребра*

**Вход**: данные ребра **data**.

**Предусловия**: нет.

**Процесс**: установка данных ребра в значение **data**.

**Выход**: нет.

**Постусловия**: данные ребра установлены в значение **data**.

*Получение данных ребра*

**Вход**: нет.

**Предусловия**: данные были установлены.

**Процесс**: обращение к полю ребра **data**.

**Выход**: данные ребра **data** или неопределённое значение, если не выполнено предусловие

**Постусловия**: нет.

*Получение веса ребра*

**Вход**: нет.

**Предусловия**: вес был установлен.

**Процесс**: обращение к полю объекта **weight**.

**Выход**: вес вершины **weight**.

**Постусловия**: нет.

*Проверка установленности веса ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **weight**.

**Выход**: true, если вес уставнолен, иначе false.

**Постусловия**: нет.

*Проверка установленности данных ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка установленности поля **data**.

**Выход**: true, если данные уставнолены, иначе false.

**Постусловия**: нет.

*Получение исходящей вершины ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: обращение к полю исходящей вершины ребра.

**Выход**: дескриптор исходящей вершины ребра.

**Постусловия**: нет.

*Получение входящей вершины ребра*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: обращение к полю входящей вершины ребра.

**Выход**: дескриптор входящей вершины ребра.

**Постусловия**: нет.

1. **Клиентское определение класса «Дескриптор ребра»**

template <class VertexT, class WeightT, class DataT>

class Edge {

VertexT \*v1, \*v2; //Вершины, которые соединяет ребро

WeightT weight; //Вес

DataT data; //Данные ребра

bool weightSet, dataSet;//признаки установки веса и данных

public:

Edge(VertexT \*v1, VertexT \*v2); //Ребро с неустановленным

//весом и данными

Edge(VertexT \*v1, VertexT \*v2, WeightT weight); //Ребро с

//установленным весом

Edge(VertexT \*v1, VertexT \*v2, WeightT weight, DataT data);

//Ребро с установленными весом и данными

void setWeight(WeightT weight); //Установка веса

void setData(DataT data); //Установка данных

WeightT getWeight(); //Получение веса

DataT getData(); //Получение данных

bool isWeightSet(); //Установленность веса

bool isDataSet();}; //Установленность данных

VertexT\* getVertex1(); //получение дескриптора исх. вершины

VertexT\* getVertex2();//получение дескриптора вх. Вершины

};

1. **Формат АТД «Итератор вершин»**

Абстрактный тип данных представляет собой вложенный в АТД «Простой граф» класс, предназначенный для последовательного доступа к вершинам графа. Итератор можно устанавливать на первую вершину графа в порядке обхода, на поледнюю вершину в порядке обхода, получать доступ к текущей вершине, а также переходить к следующей.

**ДАННЫЕ:**

*Параметры*

Текущая вершина в графе **current**.

Указатель на граф **graph**.

Признак конца просмотра **end**.

**ОПЕРАЦИИ:**

*Конструктор*

**Вход**: указатель на граф g.

**Предусловия**: нет.

**Процесс**: создание итератора вершин для графа G.

**Выход**: нет.

**Постусловия**: создан итератор вершин для графа G.

*Установка итератора на начало*

**Вход**: нет.

**Предусловия**: граф не пуст.

**Процесс**: установка итератора на первую вершину графа.

**Выход**: нет.

**Постусловия**: итератор установлен на первую вершину или не установлен при невыполненном предусловии.

*Переход к следующей вершине*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: сдвиг итератора на следующую вершину.

**Выход**: нет.

**Постусловия**: итератор установлен на следующую вершину при выполнении предусловия или в состоянии «не установлен», если предусловие не выполнено.

*Проверка неустановленности итератора*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка состояния итератора.

**Выход**: true, если итератор не установлен, иначе false.

**Постусловия**: нет.

*Доступ к текущей вершине*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: доступ к текущей вершине.

**Выход**: дескриптор вершины или исключение при невыполненном предуслов.

**Постусловия**: нет.

*Утановка итератора на конец*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: установка итератора на последнюю вершину.

**Выход**: нет.

**Постусловия**: нет.

1. **Клиентское определение класса «Итератор вершин»**

class VertexIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

bool end; //Состояние итератора

int current; //Текущее положение

public:

VertexIterator(Graph<VertexT, EdgeT> &g); //Конструктор

bool begin(); //Установка на начало

bool operator++(); //Сдвиг на след. элемент

bool onEnd(); //Проверка выхода из коллекции

VertexT\* operator\*(); //Получение доступа к вершине

void toend(); //установка на последнюю вершину

};

1. **Формат АТД «Итератор ребер»**

Абстрактный тип данных представляет собой вложенный в АТД «Простой граф» класс, предназначенный для последовательного доступа к рёбрам графа. Итератор можно устанавливать на первое ребро графа в порядке обхода, на поледнее ребро в порядке обхода, получать доступ к текущему ребру, а также переходить к следующему.

**ДАННЫЕ:**

*Параметры*

Текущее ребро графа.

Указатель на граф **graph**.

**ОПЕРАЦИИ:**

*Конструктор*

**Вход**: указатель на граф g.

**Предусловия**: нет.

**Процесс**: создание итератора рёбер для графа g.

**Выход**: нет.

**Постусловия**: создан итератор рёбер для графа g.

*Установка итератора на начало*

**Вход**: нет.

**Предусловия**: граф содержит рёбра.

**Процесс**: установка итератора на первое ребро графа.

**Выход**: **true**, если выполнено предусловие, иначе **false**.

**Постусловия**: итератор установлен на первое ребро или не установлен при невыполненном предусловии.

*Переход к следующему ребру*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: сдвиг итератора на следующее ребро.

**Выход**: **true**, если выполнено предусловие, иначе **false**.

**Постусловия**: итератор установлен на следующее ребро при выполнении предусловия или не установлен, если рёбра закончились.

*Проверка неустановленности итератора*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка состояния итератора.

**Выход**: **true**, если итератор не установлен, иначе **false**.

**Постусловия**: нет.

*Доступ к текущему ребру*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: доступ к текущему ребру.

**Выход**: дескриптор ребра или генерация исключения при невыполнении предусловия.

**Постусловия**: нет.

*Утановка итератора на конец*

**Вход**: нет.

**Предусловия**: проверка наличия рёбер.

**Процесс**: установка итератора на последнее ребро.

**Выход**: **true**, если итератора установлен на последнее ребро, **false** – при невыполнении предусловия.

**Постусловия**: нет.

1. **Клиентское определение класса «Итератор ребер»**

class EdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

bool end; //Состояние итератора

int curV1, curV2; //Текущее положение

public:

EdgeIterator(Graph<VertexT, EdgeT> &g); //Конструктор

bool begin(); //Установка на начало

bool operator++(); //Сдвиг на след. элемент

bool onEnd(); //Проверка выхода из коллекции

EdgeT\* operator\*(); //Получение доступа к ребру

bool toend(); //установка на последнее ребро

};

1. **Формат АТД «Итератор исходящих ребер»**

Абстрактный тип данных представляет собой вложенный в АТД «Простой граф» класс, предназначенный для последовательного доступа к исходящим рёбрам вершин графа. Итератор можно устанавливать на первое исходящее ребро вершины графа в порядке обхода, на поледнее исходящее ребро вершины графа в порядке обхода, получать доступ к текущему ребру вершины графа, а также переходить к следующему.

**ДАННЫЕ:**

Текущее ребро графа.

Указатель на граф **graph**.

Номер рассматриваемой вершины **curV1**.

**ОПЕРАЦИИ:**

*Конструктор*

**Вход**: указатель на граф G и дескриптор вершины V.

**Предусловия**: проверка наличия вершины V в графе.

**Процесс**: создание итератора исходящих рёбер для вершины V графа G.

**Выход**: нет.

**Постусловия**: создан итератор рёбер для вершины V графа G, либо генерация исключения при невыполнеии предусловия.

*Установка итератора на начало*

**Вход**: нет.

**Предусловия**: вершина содержит исходящие рёбра.

**Процесс**: установка итератора на первое исходящее ребро вершины.

**Выход**: **true**, если выполнено предусловие, иначе **false**.

**Постусловия**: итератор установлен на первое исходящее ребро.

*Переход к следующему исходящему ребру*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: сдвиг итератора на следующее исходящее ребро.

**Выход**: **true**, если выполнено предусловие, иначе **false**.

**Постусловия**: итератор установлен на следующее исходящее ребро при вып. предусловии или не установлен, если рёбра закончились.

*Проверка неустановленности итератора*

**Вход**: нет.

**Предусловия**: нет.

**Процесс**: проверка состояния итератора.

**Выход**: **true**, если итератор не установлен, иначе **false**.

**Постусловия**: нет.

*Доступ к текущему ребру:*

**Вход**: нет.

**Предусловия**: итератор установлен.

**Процесс**: доступ к текущему ребру.

**Выход**: дескриптор исходящего ребра или генерация исключения при невыполнении предусловия.

**Постусловия**: нет.

*Утановка итератора на конец*

**Вход**: нет.

**Предусловия**: проверка наличия исходящих рёбер.

**Процесс**: установка итератора на последнее исходящее ребро.

**Выход**: **true**, если итератора установлен на последнее исходящее ребро, **false** – при невыполнении предусловия.

**Постусловия**: нет

1. **Клиентское определение класса «Итератор исходящих ребер»**

class OutputEdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

bool end; //Состояние итератора

int curV1, curV2; //Текущее положение

public:

OutputEdgeIterator (Graph<VertexT, EdgeT> &g, VertexT &v); //Конструктор

bool begin(); //Установка на начало

bool operator++(); //Сдвиг на след. элемент

bool onEnd(); //Проверка выхода из коллекции

EdgeT\* operator\*();//Получение доступа к исходящему ребру

bool toend(); //установка на последнее исходящее ребро

};

1. **Формат АТД «Задача 1»**

Абстрактный тип данных представляет собой класс, формирующий списк ребер неориентированного графа в порядке двухпроходного эйлерова цикла.

**ДАННЫЕ:**

*Параметры*

Обрабатываемый граф **graph**.

Переменная, содержащая список ребер **st**.

**ОПЕРАЦИИ:**

*Конструктор:*

**Вход:** граф для решения задачи.

**Предусловия:** нет.

**Процесс:** решение задачи для данного графа.

**Выход:** нет.

**Постусловия:** построен граф.

*Конструктор копирования:*

**Вход:** граф для решения задачи.

**Предусловия:** нет.

**Процесс:** создание копии объекта задачи.

**Выход:** нет.

**Постусловия:** создана копия объекта задачи.

*Поиск в глубину для заданной вершины:*

**Вход:** вершина с исходящим ребром.

**Предусловия:** нет.

**Процесс:** нет.

**Выход:** список пройденных вершин.

**Постусловия:** нет.

*Получение списка смежных вершин:*

**Вход:** вершина с исходящим ребром.

**Предусловия:** нет.

**Процесс:** определение смежных вершин.

**Выход:** список смежных вершин.

**Постусловия:** список вершин.

*Получение решения:*

**Вход:** нет.

**Предусловия:** нет.

**Процесс:** проход по всем ребрам.

**Выход:** список пройденных ребер.

**Постусловия:** нет.

1. **Клиентское определение класса «Задача 1»**

class Euler {

Euler(Graph<VertexT, EdgeT> &g); //конструктор

Euler(const Graph<VertexT, EdgeT> &T); //конструктор копирования

~Euler(); //деструктор

bool DFS (); //поиск в глубину пока возможно

void search (); //получаем списоки смежности для каждой вершины

void DFS\_Visit (int u); //поиск для заданной вершины

};

1. **Формат АТД «Задача 2»**

Абстрактный тип данных представляет собой класс, определяющий центр взвешенного орграфа на основе алгоритма Беллмана – Форда.

**ДАННЫЕ:**

*Параметры*

Обрабатываемый граф **graph**.

**ОПЕРАЦИИ:**

*Конструктор:*

**Вход:** граф для решения задачи.

**Предусловия:** нет.

**Процесс:** решение задачи для данного графа.

**Выход:** нет.

**Постусловия:** построен граф.

*Конструктор копирования:*

**Вход:** граф для решения задачи.

**Предусловия:** нет.

**Процесс:** создание копии объекта задачи.

**Выход:** нет.

**Постусловия:** создана копия объекта задачи.

*Получение решения:*

**Вход:** список смежных вершин.

**Предусловия:** нет

**Процесс:** определение центра данного графа.

**Выход:** если отрицательных циклов нет - центр данного графа, иначе отрицательный цикл.

**Постусловия:** нет.

1. **Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 1**

Стратегия поиска в глубину (Depth - First Search - DFS) следующая: идти "вглубь" пока это возможно, возвращаться и искать другой путь, когда таких ребер нет. Если после этого остаются необнаруженные вершины, можно выбрать одну из них и повторить процесс и делать так до тех пор, пока не будут обнаружены все вершины граф.

Для вершины используется поле color[u], задающее БЕЛЫЙ, СЕРЫЙ или ЧЕРНЫЙ цвет. В начале все вершины белые. В процессе поиска при обнаружении вершина становится серой. Вершина становится черной, когда она полностью обработана, т. е. когда список смежных с ней вершин полностью просмотрен. Каждая вершина попадает только в одно дерево поиска в глубину, и эти деревья не пересекаются.

***Трудоемкость алгоритма:***

Циклы в функции DFS 1 - 3 и 5 - 7 требуют O(V) времени. Функция DFS - Visit вызывается один раз для каждой вершины. В функции DFS - Visit цикл 3 - 6 выполняется Adj[V] раз.

По всем вершинам графа общее время функции DFS - Visit оценивается как O[E].

Таким образом общее время обхода в глубину оценивается, как O(V + E).

1. **Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 2**

Алгоритм Беллмана-Форда решает задачу о кратчайших путях из одной вершины для случая, когда веса ребер могут быть отрицательными. Алгоритм возвращает значение TRUE если в кратчайших путях нет отрицательного цикла и FALSE, если такой цикл есть. В первом случае алгоритм находит кратчайшие пути и их веса, во втором случае - кратчайших путей нет.

После инициализации вершин алгоритм |V| - 1 раз перебирает по все ребра и подвергает каждое из них релаксации.

Количество итераций равно |V| - 1, так как на кратчайших путях не может быть более |V| - 1 ребер.

После релаксаций алгоритм проверяет нет ли цикла отрицательного веса, достижимого из исходной вершины. Признаком этого является наличие на кратчайшем пути ребер (u, v), для которых d[v]> d[u] + w (u, v).

***Трудоемкость алгоритма:*** O(VE).

1. **Заключение**

В результате проделанной работы был разработан АТД «Простой граф», а также ассоциированные с ним классы, реализующие шаблонные типы данных. Коллекция повзоляет реализовывать ориентированные и неориентированные, взвешенные и невзвешенные графы M и L типов.

На разработанном графе были реализованы и протестированы задачи, связанные с графом, такие как двухпроходной Эйлеров цикл и поиск центра графа на основе алгоритма Белмана - Форда.

1. **Список использованных источников**
2. С/С++. Программирование на языке высокого уровня: учебник для вузов. Т.А. Павловская. – Санкт-Петербург: Изд-во Питер, 2006.
3. Абстракция данных и решение задач на С++. Стены и зеркала. Ф. Каррано, Д. Причард. – Москва: издательский дом «Вильямс», 2003.
4. Структуры данных и алгоритмы: учебное пособие. Альфред В. Ахо, Джон Э. Хопкрофт, Джеффри Д. Ульман. – Москва: Издательский дом «Вильямс», 2000.
5. **Приложения**

Graph.h

#include "iostream"

#include "conio.h"

#include "string"

#include "vector"

#include "list"

#include "iomanip"

#include <sstream>

#include <stack>

using namespace std;

int sz=0;

string i\_to\_s(int x){

char buffer[20];

\_itoa(x, buffer, 10);

string s = buffer;

return s;

}

//АТД «Дескриптор вершины графа»

template <class NameT, class DataT> class Vertex {

NameT name; //Имя вершины

DataT data; //Данные вершины

public:

bool nameSet, dataSet;

//Конструкторы

Vertex() :

nameSet(false),

dataSet(false) {}

Vertex(NameT name, DataT data) :

name(name),

data(data),

nameSet(true),

dataSet(true) {}

//Установка, получение имени и данных

void setName(NameT name) {

this->name = name;

nameSet = true;

}

void setData(DataT data) {

this->data = data;

dataSet = true;

}

NameT getName() {

return name;

}

DataT getData() {

return data;

}

bool isNameSet() {

return nameSet;

}

bool isDataSet() {

return dataSet;

}

};

//АТД «Дескриптор ребра графа»

template <class VertexT, class WeightT, class DataT> class Edge {

VertexT \*v1, \*v2; //Вершины, которые соединяет ребро

WeightT weight; //Вес

DataT data; //Данные ребра

bool weightSet, dataSet;

public:

//Конструкторы

Edge(VertexT \*v1, VertexT \*v2) :

v1(v1),

v2(v2),

weightSet(false),

weight(0),

dataSet(false) {}

Edge(VertexT \*v1, VertexT \*v2, WeightT weight) :

v1(v1),

v2(v2),

weight(weight),

weightSet(true),

dataSet(false) {}

Edge(VertexT \*v1, VertexT \*v2, WeightT weight, DataT data) :

v1(v1),

v2(v2),

weight(weight),

data(data),

weightSet(true),

dataSet(true) {}

//Установка и получение веса и данных

void setWeight(WeightT weight) {

this->weight = weight;

weightSet = true;

}

void setData(DataT data) {

this->data = data;

dataSet = true;

}

WeightT getWeight() {

return weight;

}

DataT getData() {

return data;

}

VertexT\* getVertex1(){

return v1;

};

VertexT\* getVertex2(){

return v2;

};

bool isWeightSet() {

return weightSet;

}

bool isDataSet() {

return dataSet;

}

};

//форма представления

template <class EdgeT> class GraphForm {

public:

//Вставка и удаление вершин и рёбер

virtual bool InsertV(int index) = 0;

virtual bool DeleteV(int index) = 0;

virtual bool InsertE(int v1, int v2, EdgeT \*t) = 0;

virtual bool DeleteE(int v1, int v2) = 0;

//Удалить входящие и исходящие из вершины рёбра

virtual int DeleteEsFromVertex(int index, bool directed) = 0;

//Проверка и получение

virtual bool hasEdge(int v1, int v2) = 0;

virtual EdgeT\* getEdge(int v1, int v2) = 0;

};

//форма представления матричная

template <class EdgeT> class GraphMatrixForm : public GraphForm<EdgeT> {

friend class EdgeIterator;

bool directed;

public:

vector<vector<EdgeT\*>> matrix; //Матрица смежности

vector<vector<EdgeT\*>> matrix\_tmp1; //temp Матрица смежности

vector<vector<EdgeT\*>> matrix\_tmp2; //temp Матрица смежности

GraphMatrixForm(bool directed) : directed(directed) {}

//Вставка и удаление вершин и рёбер

bool InsertV(int index) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(index < 0 || index > size) return false;

//Создаём новую пустую строку

vector<EdgeT\*> newLine;

newLine.assign(size, NULL);

//Вставляем новую строку:

matrix.insert(matrix.begin() + index, newLine);

++size;

//Вставляем новый столбец:

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

matrix[i].insert(matrix[i].begin() + index, (EdgeT\*)NULL);

return true;

}

bool DeleteV(int index) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(index < 0 || index >= size) return false;

//Удаляем строку:

matrix.erase(matrix.begin() + index);

--size;

//Удаляем столбец:

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

matrix[i].erase(matrix[i].begin() + index);

return true;

}

bool InsertE(int v1, int v2, EdgeT \*t) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size) return false;

//Петля или ребро уже есть

if(v1 == v2 || matrix[v1][v2] != NULL) return false;

//Вставляем ребро

matrix[v1][v2] = t;

return true;

}

bool DeleteE(int v1, int v2) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size) return false;

//Ребра нет

if(v1 == v2 || matrix[v1][v2] == NULL) return false;

matrix[v1][v2] = NULL;

return true;

}

//Удалить входящие и исходящие из вершины рёбра

int DeleteEsFromVertex(int index, bool directed) {

int size = matrix.size(); //Число вершин

int deleted = 0;

//Неверный номер вершины

if(index < 0 || index >= size) return 0;

//Удаляем связанные с вершиной рёбра

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

if(matrix[i][index] != NULL) {

delete matrix[i][index];

matrix[i][index] = NULL;

++deleted;

//Стираем симметричное ребро

if(directed == false)

matrix[index][i] = NULL;

}

if(matrix[index][i] != NULL) {

delete matrix[index][i];

matrix[index][i] = NULL;

++deleted;

}

}

return deleted;

}

//Проверка и получение

bool hasEdge(int v1, int v2) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

return false;

if(v1 == v2) //Петля

return false;

if(matrix[v1][v2] != NULL)

return true;

return false;

}

EdgeT\* getEdge(int v1, int v2) {

int size = matrix.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

throw 1;

if(v1 == v2 || matrix[v1][v2] == NULL)//Петля

throw 1;

return matrix[v1][v2];

}

//Итератор рёбер

class EdgeIterator {

bool end;

GraphMatrixForm \*f;

public:

int curI, curJ;

EdgeIterator(GraphMatrixForm \*f) {

this->f = f;

begin();

}

bool begin() {

for(int i = 0; i < f->matrix.size(); ++i)

for(int j = 0; j < f->matrix.size(); ++j)

if(f->matrix[i][j]) {

curI = i;

curJ = j;

end = false;

return true;

}

end = true;

return false;

}

bool onEnd() {

return end;

}

bool toend(){

for(int i=f->matrix.size()-1; i>=0; --i)

for(int j=f->matrix.size()-1; j>=0; --j)

if (f->matrix[i][j]){

curI=i;

curJ=j;

end=false;

return true;

}

end=true;

return false;

}

bool next() {

if(end)

return false;

++curJ;

while(curI < f->matrix.size()) {

while(curJ < f->matrix.size()) {

if(f->matrix[curI][curJ])

return true;

++curJ;

}

++curI;

curJ = (f->directed ? 0 : curI + 1);

}

end = true;

return false;

}

EdgeT\* getEdge() {

if(end)

throw 1;

if (!(f->matrix[curI][curJ]))

throw 1;

return f->matrix[curI][curJ];

}

};

//Итератор исходящих рёбер

class OutputEdgeIterator {

int curI, curJ;

bool end;

GraphMatrixForm \*f;

public:

OutputEdgeIterator(GraphMatrixForm \*f, int vIndex) {

this->f = f;

curI = vIndex;

begin();

}

bool begin() {

for(curJ = 0; curJ < f->matrix.size(); ++curJ)

if(f->matrix[curI][curJ]) {

end = false;

return true;

}

end = true;

return false;

}

bool toend(){

for(curJ=f->matrix.size()-1; curJ>=0; --curJ)

if (f->matrix[curI][curJ]){

end=false;

return true;

}

end=true;

return false;

}

bool onEnd() {

return end;

}

bool next() {

if(end)

return false;

++curJ;

while(curJ < f->matrix.size()) {

if(f->matrix[curI][curJ])

return true;

++curJ;

}

end = true;

return false;

}

EdgeT\* getEdge() {

if(end)

throw 1;

return f->matrix[curI][curJ];

}

};

};

//форма представления список

template <class EdgeT> class GraphListForm : public GraphForm<EdgeT> {

//Элемент списка

class Node {

public:

EdgeT \*edge; //Само ребро

int v2; //Вторая вершина, которую ребро соединяет

};

bool directed;

vector<list<Node>> edgeList; //Списки смежности

public:

GraphListForm(bool directed) : directed(directed) {}

//Вставка и удаление вершин и рёбер

bool InsertV(int index) {

int size = edgeList.size(); //Число вершин

if(index < 0 || index > size) //Неверный номер вершины

return false;

//Создаём новый список смежности

list<Node> newList;

//Вставляем

edgeList.insert(edgeList.begin() + index, newList);

++size;

//Обновляем дескрипторы

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 >= index)//если текущая вершина имеет больший номер, чем вставляемая,

++((\*j).v2);//то увеличиваем этот номер

return true;

}

bool DeleteV(int index) {

int size = edgeList.size(); //Число вершин

if(index < 0 || index >= size) //Неверный номер вершины

return false;

//Удаляем из списков записи о рёбрах

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 == index) {

edgeList[i].erase(j);

break;

}

//Удаляем список смежности

edgeList.erase(edgeList.begin() + index);

--size;

//Обновляем дескрипторы

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 > index)//если текущая вершина имеет больший номер, чем удаляемая,

--((\*j).v2);//то уменьшить этот номер

return true;

}

bool InsertE(int v1, int v2, EdgeT \*t) {

int size = edgeList.size(); //Число вершин

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)//Неверный номер вершины

return false;

if(v1 == v2 || hasEdge(v1, v2)) //Петля или ребро уже есть

return false;

//Вставляем ребро

Node node;

node.edge = t;

node.v2 = v2;

edgeList[v1].push\_front(node);

return true;

}

bool DeleteE(int v1, int v2) {

int size = edgeList.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

return false;

//Ребра нет

if(v1 == v2 || hasEdge(v1, v2) == false)

return false;

//Удаляем ребро

for(list<Node>::iterator j = edgeList[v1].begin(); j != edgeList[v1].end(); ++j)

if((\*j).v2 == v2) {

edgeList[v1].erase(j);

break;

}

return true;

}

//Удалить входящие и исходящие из вершины рёбра

int DeleteEsFromVertex(int index, bool directed) {

int size = edgeList.size(); //Число вершин

int deleted = 0;

//Неверный номер вершины

if(index < 0 || index >= size)

return 0;

//Удаляем связанные с вершиной рёбра

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

for(list<Node>::iterator j = edgeList[i].begin(); j != edgeList[i].end(); ++j)

if((\*j).v2 == index) {

delete (\*j).edge;

edgeList[i].erase(j);

++deleted;

//Стираем симметричное ребро

if(directed == false)

for(list<Node>::iterator k = edgeList[index].begin(); k != edgeList[index].end(); ++k)

if((\*k).v2 == i) {

edgeList[index].erase(k);

break;

}

break;

}

if (directed==true){

for(list<Node>::iterator z = edgeList[index].begin(); z != edgeList[index].end(); ++z){

delete (\*z).edge;

deleted++;

}

}

return deleted;

}

//Проверка и получение

bool hasEdge(int v1, int v2) {

int size = edgeList.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

return false;

//Петля

if(v1 == v2)

return false;

for(list<Node>::iterator j = edgeList[v1].begin(); j != edgeList[v1].end(); ++j)

if((\*j).v2 == v2)

return true;

return false;

}

EdgeT\* getEdge(int v1, int v2) {

int size = edgeList.size(); //Число вершин

//Неверный номер вершины

if(v1 < 0 || v2 < 0 || v1 >= size || v2 >= size)

throw 1;

//Петля

if(v1 == v2)

throw 1;

for(list<Node>::iterator j = edgeList[v1].begin(); j != edgeList[v1].end(); ++j)

if((\*j).v2 == v2)

return (\*j).edge;

throw 1;

}

//Итератор рёбер

class EdgeIterator {

int curI;

typename list<typename GraphListForm<EdgeT>::Node>::iterator curJ;

bool end;

GraphListForm \*f;

public:

EdgeIterator(GraphListForm \*f) {

this->f=f;

}

//Установить итератор рёбер на начальное ребро

bool begin() {

for(curI = 0; curI < f->edgeList.size(); ++curI)

for(curJ = f->edgeList[curI].begin(); curJ != f->edgeList[curI].end(); ++curJ)

if((\*curJ).edge) {

end = false;

return true;

}

end = true;

return false;

}

bool onEnd() {

return end;

}

//Установить итератор рёбер на конечное ребро

bool toend(){

typename list<typename GraphListForm<EdgeT>::Node>::iterator prev\_curJ;

for(curI = 0; curI < f->edgeList.size(); ++curI) {

for(curJ = f->edgeList[curI].begin(); curJ != f->edgeList[curI].end(); ++curJ) {

if ((\*curJ).edge)

prev\_curJ=curJ;

if (!(\*prev\_curJ).edge){

end=true;

return false;

}

else{

end=false;

curJ=prev\_curJ;

return true;

}

}

}

};

//Переход к следующей позиции

bool next() {

if(end)

return false;

++curJ;

while(curI < f->edgeList.size()) {

while(curJ != f->edgeList[curI].end()) {

if((\*curJ).edge && (f->directed || !f->directed && (\*curJ).v2 > curI))

return true;

++curJ;

}

++curI;

if(curI < f->edgeList.size())

curJ = f->edgeList[curI].begin();

}

end = true;

return true;

}

EdgeT\* getEdge() {

if(end)

throw 1;

return (\*curJ).edge;

}

};

//Итератор исходящих рёбер

class OutputEdgeIterator {

int curI;

typename list<typename GraphListForm<EdgeT>::Node>::iterator curJ;

bool end;

GraphListForm \*f;

public:

OutputEdgeIterator(GraphListForm \*f, int vIndex) {

this->f = f;

curI = vIndex;

begin();

}

bool begin() {

for(curJ = f->edgeList[curI].begin(); curJ != f->edgeList[curI].end(); ++curJ)

if((\*curJ).edge) {

end = false;

return true;

}

end = true;

return false;

}

bool onEnd() {

return end;

}

bool toend(){

typename list<typename GraphListForm<EdgeT>::Node>::iterator prev\_curJ;

for(curJ = f->edgeList[curI].begin(); curJ != f->edgeList[curI].end(); ++curJ)

if ((\*curJ).edge)

prev\_curJ=curJ;

if (!(\*prev\_curJ).edge){

end=true;

return false;

}

else{

end=false;

curJ=prev\_curJ;

return true;

};

}

bool next() {

if(end)

return false;

++curJ;

while(curJ != f->edgeList[curI].end()) {

if((\*curJ).edge)

return true;

++curJ;

}

end = true;

return true;

}

EdgeT\* getEdge() {

if(end)

throw 1;

return (\*curJ).edge;

}

};

};

//АТД простой граф

template <class VertexT, class EdgeT> class Graph {

friend class VertexIterator; //дружественный класс

vector<VertexT\*> vertexVector;//Вектор вставленных

GraphForm<EdgeT> \*data; //Данные графа

bool directed; //Направленный граф

bool dense; //М-граф

int edgeCounter; //Число рёбер

//Получение индекса вершины по дескриптору

int getIndexFromName(string n){

int index=0;

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

if (vertexVector[index]->getName()==n)

break;

++index;

}

if(index==vertexVector.size())

throw "Exeption#";

return index;

};

public:

//Создать пустой L-граф с нулевым числом вершин и рёбер

Graph() {

data = new GraphListForm<EdgeT>(false);

directed = false;

dense = false;

edgeCounter = 0;

}

//Создать граф с V вершинами, без ребер, типа D, формы представления F

Graph(int vertexCount, bool directed, bool dense) {

if(dense)

data = new GraphMatrixForm<EdgeT>(directed);//!!!!!!!!!!!!1

else

data = new GraphListForm<EdgeT>(directed);//!!!!!!!!!!!!!!!!!!!!

//Создаём вершины и помещаем в структуру

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

data->InsertV(i);

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

VertexT \*v = new VertexT();

v->setName(i\_to\_s(sz));

sz++;

vertexVector.push\_back(v);

data->InsertV(i);

}

this->directed = directed;

this->dense = dense;

edgeCounter = 0;

}

//Создать граф с V вершинами, с E случайными ребрами, типа D , формы F

Graph(int vertexCount, int edgeCount, bool directed, bool dense) {

if(dense)

data = new GraphMatrixForm<EdgeT>(directed);

else

data = new GraphListForm<EdgeT>(directed);

//Создаём вершины и помещаем в структуру

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

data->InsertV(i);

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

VertexT \*v = new VertexT();

v->setName(i\_to\_s(sz));

sz++;

vertexVector.push\_back(v);

data->InsertV(i);

}

this->directed = directed;

this->dense = dense;

edgeCounter = 0;

//Вставляем рёбра

if(edgeCount <= 0) return;

//Корректируем число рёбер, чтобы не превышало максимумы

if(directed) {

if(edgeCount > vertexCount \* (vertexCount - 1))

edgeCount = vertexCount \* (vertexCount - 1);

srand(time(0));

} else {

if(edgeCount > (vertexCount \* (vertexCount - 1)) / 2)

edgeCount = (vertexCount \* (vertexCount - 1)) / 2;

}

//Вставляем

int v1, v2;

while(edgeCounter < edgeCount) {

v1 = rand() % vertexCount;

v2 = rand() % vertexCount;

if(v1 == v2)

continue;

if(data->hasEdge(v1, v2))

continue;

EdgeT \*t = new EdgeT(vertexVector[v1], vertexVector[v2]);

data->InsertE(v1, v2, t);

t->setWeight((rand() % 9) +1);

if(directed == false)

data->InsertE(v2, v1, t);

++edgeCounter;

}

}

//конструктор копирования

Graph(const Graph<VertexT, EdgeT> &G) {

delete data;

if(G.Dense())

data = new GraphMatrixForm<EdgeT>(\*(G.data));

else

data = new GraphListForm<EdgeT>(\*(G.data));

directed = G.directed;

dense = G.dense;

}

//деструктор

~Graph() {

while (vertexVector.size()!=0)

DeleteV(vertexVector.size()-1);

delete data;

}

//возвращает число вершин в графе

int V() {

return vertexVector.size();

}

//возвращает число ребер в графе

int E() {

return edgeCounter;

}

//возвращает тип графа

bool Directed() {

return directed;

}

//возвращает форму представления графа

bool Dense() {

return dense;

}

//возвращает коэффициент насыщенности графа

double K() {

int max = vertexVector.size() \* (vertexVector.size() - 1);

if(!directed) {

max /= 2;

}

if (max==0) {

return -1;

}

else {

return (double)edgeCounter / (double)max;

}

}

//преобразует граф к L-графу

void ToListGraph() {

//Создаём структуру хранения L-графа

GraphForm<EdgeT> \*newData = new GraphListForm<EdgeT>(this->directed);

//Создаём вершины

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

newData->InsertV(i);

//Переносим рёбра

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

for(int j = 0; j < vertexVector.size(); ++j)

if(data->hasEdge(i, j))

newData->InsertE(i, j, data->getEdge(i, j));

delete data;

data = newData;

dense = false;

}

//преобразует граф к M-графу

void ToMatrixGraph() {

//Создаём структуру хранения М-графа

GraphForm<EdgeT> \*newData = new GraphMatrixForm<EdgeT>(this->directed);

//Создаём вершины

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

newData->InsertV(i);

//Переносим рёбра

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

for(int j = 0; j < vertexVector.size(); ++j)

if(data->hasEdge(i, j))

newData->InsertE(i, j, data->getEdge(i, j));

delete data;

data = newData;

dense = true;

}

//добавляет вершину к графу и возвращает адрес дескриптора вновь созданной вершины

VertexT\* InsertV() {

VertexT \*v = new VertexT;

if(data->InsertV(vertexVector.size()) == false)

throw 1;

v->setName(i\_to\_s(sz));

sz++;

vertexVector.push\_back(v);

return v;

}

//удаляет вершину из графа, заданную адресом дескриптора v

bool DeleteV(VertexT \*v) {

int index = getIndex(v);

edgeCounter -= data->DeleteEsFromVertex(index, directed);

if(data->DeleteV(index)) {

vertexVector.erase(vertexVector.begin() + index);

return true;

}

return false;

}

bool DeleteV(int index) {

edgeCounter -= data->DeleteEsFromVertex(index, directed);

if(data->DeleteV(index)) {

vertexVector.erase(vertexVector.begin() + index);

return true;

}

return false;

}

bool DeleteV(string n) {

VertexT \*v=new VertexT;

bool rez;

try{

v=getVertexFromName(n);

}

catch(char\*){

return false;

}

rez=DeleteV(v);

return rez;

}

VertexT\* getVertex(int index) {

if(index < 0 || index >= vertexVector.size())

throw "Exeption#";

return vertexVector[index];

}

VertexT\* getVertexFromName(string n){

int i;

for(i=0; i<vertexVector.size(); i++)

if (vertexVector[i]->getName()==n)

return vertexVector[i];

if (i==vertexVector.size())

throw "Exeption#";

};

int getIndex(VertexT \*v) {

int index = 0;

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

if(vertexVector[index] == v)

break;

++index;

}

if(index == vertexVector.size())

throw "Exeption#";

return index;

}

//добавляет ребро (v1, v2) к графу, соединяющую вершины, заданные адресами дескрипторов v1 и v2

bool InsertE(string v1, string v2){

VertexT\* \_v1=new VertexT;

VertexT\* \_v2=new VertexT;

try{

\_v1=getVertexFromName(v1);

}

catch(char\* e){

return false;

};

try{

\_v2=getVertexFromName(v2);

}

catch(char\* e){

return false;

}

try{

InsertE(\_v1, \_v2);

}

catch (char\* e){

return false;

}

return true;

};

EdgeT\* InsertE(VertexT \*v1, VertexT \*v2) {

EdgeT \*e = new EdgeT(v1, v2);

if(!data->InsertE(getIndex(v1), getIndex(v2), e)) throw "Exeption#";

if(directed == false)

data->InsertE(getIndex(v2), getIndex(v1), e);

++edgeCounter;

return e;

}

//удаляет ребро, соединяющее вершины, заданные адресами дескрипторов v1 и v2

bool DeleteE(string v1, string v2){

VertexT\* \_v1=new VertexT;

VertexT\* \_v2=new VertexT;

try{

\_v1=getVertexFromName(v1);

}

catch(char\*e){

//cerr<<endl<<e<<endl;

return false;

};

try{

\_v2=getVertexFromName(v2);

}

catch(char\*e){

//cerr<<endl<<e<<endl;

return false;

}

if(!DeleteE(\_v1, \_v2)) return false;

else return true;

};

bool DeleteE(VertexT \*v1, VertexT \*v2) {

if(data->DeleteE(getIndex(v1), getIndex(v2))) {

--edgeCounter;

if(directed == false)

data->DeleteE(getIndex(v2), getIndex(v1));

return true;

} else

return false;

}

bool hasEdge(VertexT \*v1, VertexT \*v2) {

int ind1, ind2;

try {

ind1 = getIndex(v1);

}

catch(int &a) {

return false;

}

try {

ind2 = getIndex(v2);

}

catch(int &a)

{

return false;

}

return data->hasEdge(getIndex(v1), getIndex(v2));

}

bool hasEdge(int v1, int v2) {

if(v1 < 0 || v1 >= vertexVector.size())

return false;

if(v2 < 0 || v2 >= vertexVector.size())

return false;

return data->hasEdge(v1, v2);

}

EdgeT\* getEdge(VertexT \*v1, VertexT \*v2) {

EdgeT\* e;

try{

e=data->getEdge(getIndex(v1), getIndex(v2));

}

catch(int){

throw "Exeption#";

}

return e;

}

//Записать данные о ребре

bool write\_data\_edge(string \_v1, string \_v2, int data){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

}

catch(char\*){

return false;

}

try{

v2=getVertexFromName(\_v2);

}

catch(char\*){

return false;

}

try {

e=getEdge(v1, v2);

e->setData(data);

}

catch (char\*e) {

cerr<<endl<<e<<endl;

return false;

}

return true;

};

//Записать вес ребра

bool write\_weight\_edge(string \_v1, string \_v2, int w){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

}

catch(char\*){

return false;

}

try{

v2=getVertexFromName(\_v2);

}

catch(char\*){

return false;

}

try {

e=getEdge(v1, v2);

e->setWeight(w);

}

catch (char\*e) {

cerr<<endl<<e<<endl;

return false;

}

return true;

};

//Прочитать данные о ребре

int read\_data\_edge(string \_v1, string \_v2){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

v2=getVertexFromName(\_v2);

}

catch(char\*e){

cerr<<endl<<e<<endl;

return -1;

}

try {

e=getEdge(v1, v2);

}

catch (char\*e) {

cerr<<endl<<e<<endl;

return -1;

}

try {

if (e->isDataSet())

return e->getData();

else

throw "Exeption#";

}

catch (char\*e) {

cerr<<endl<<e<<endl;

return -1;

}

};

//Прочитать вес ребра

int read\_weight\_edge(string \_v1, string \_v2){

EdgeT\* e;

VertexT \*v1, \*v2;

try{

v1=getVertexFromName(\_v1);

v2=getVertexFromName(\_v2);

e=getEdge(v1, v2);

}

catch(char\*e){

cerr<<endl<<e<<endl;

return -1;

}

try {

if (e->isWeightSet())

return e->getWeight();

else

throw "Exeption#";

}

catch (char\*e) {

cerr<<endl<<e<<endl;

return -1;

}

};

//Вывод

void print\_graph(){

int i, j;

VertexT\* v;

EdgeT\* e;

if (Dense()){

cout<<" ";

for (i=0; i<V(); i++){

v=getVertex(i);

cout<<setw(4)<<v->getName();

}

cout<<endl;

for(i=0; i<5\*V(); i++)

cout<<"\_";

cout<<endl;

for (i=0; i<V(); i++){

v=getVertex(i);

cout<<v->getName()<<"|";

for (j=0; j<V(); j++)

if (hasEdge(i,j)){

e=getEdge(getVertex(i),getVertex(j));

cout<<setw(4)<<e->getWeight();

}

else

cout<<setw(4)<<"0";

cout<<endl;

}

}

else

{

for (i=0; i<V(); i++){

v=getVertex(i);

cout<<"\*"<<v->getName()<<"->";

for (j=0; j<V(); j++){

v=getVertex(j);

if (hasEdge(i,j))

cout<<v->getName()<<"->";

}

cout<<endl;

}

}

};

//Прочитать данные о вершине

int read\_data\_vertex(string v1){

VertexT \*v;

try

{

v=getVertexFromName(v1);

}

catch(char\* e){

cerr<<endl<<e<<endl;

return -1;

};

int d;

d=v->getData();

try {

if (v->dataSet!=false)

return d;

else

throw invalid\_argument ("Exeption#");

}

catch (invalid\_argument &e) {

cerr<<endl<<e.what()<<endl;

return -1;

}

};

//Записать данные о вершине

bool write\_data\_vertex(string v1, int d){

VertexT \*v;

try

{

v=getVertexFromName(v1);

}

catch(char\*){

return false;

};

v->setData(d);

return true;

};

//Записать данные о вершине

bool write\_name\_vertex(string v1, string str){

VertexT \*v;

try

{

v=getVertexFromName(v1);

}

catch(char\*){

return false;

};

v->setName(str);

return true;

};

//Итератор вершин

class VertexIterator {

Graph<VertexT, EdgeT> \*graph;

bool end;

int current;

public:

VertexIterator(Graph<VertexT, EdgeT> &g) {

graph = &g;

//begin();

}

//возвращает итератор, установленный на первую вершину графа

bool begin() {

if(graph->vertexVector.size() == 0) {

end = true;

return true;

}

current = 0;

end = false;

return true;

}

//Переход к следующей позиции

bool operator++() {

try {

if(end){

current=-1;

throw "Exeption#";

}

}

catch (char\*e) {

cerr<<endl<<e<<endl;

return false;

}

current++;

if(graph->vertexVector.size()-1 == current)

end = true;

return true;

};

//Установить итератор вершин на конечную вершину

bool toend(){

current=graph->vertexVector.size()-1;

end=true;

return true;

};

//Проверка на выход

bool onEnd() {

return end;

}

//Получение вершины

VertexT\* operator\*() {

if(current >= graph->vertexVector.size() || current==-1)

throw "Exeption#";

return graph->vertexVector[current];

}

//Прочитать вершину по текущей позиции итератора

string read\_vertex() {

VertexT \*v;

try

{

v=operator\*();

}

catch(char\*){

throw "Exeption#";

};

string d;

d=v->getName();

if (v->nameSet!=false)

return d;

else

throw "Exeption#";

}

//Прочитать данные вершины по текущей позиции итератора

int read\_data\_vertex(){

VertexT \*v;

try

{

v=operator\*();

}

catch(char\*e){

cerr<<endl<<e<<endl;

return -1;

};

try {

if (v->dataSet!=false)

return v->getData();

else throw "Exeption#";

}

catch (char\*e) {

cerr<<endl<<e<<endl;

return -1;

}

};

//Записать данные о вершины по текущей позиции итератора

bool write\_data\_vertex(int d){

VertexT \*v;

try

{

v=operator\*();

}

catch(char\*e){

cerr<<endl<<e<<endl;

return false;

};

v->setData(d);

return true;

};

//Записать имя вершины по текущей позиции итератора

bool write\_name\_vertex(string str){

VertexT \*v;

try

{

v=operator\*();

}

catch(char\*e){

cerr<<endl<<e<<endl;

return false;

};

v->setName(str);

return true;

};

};

//Итератор рёбер

class EdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

typename GraphMatrixForm<EdgeT>::EdgeIterator \*mIt;

typename GraphListForm<EdgeT>::EdgeIterator \*lIt;

bool useM;

public:

int curV1, curV2;

string str1, str2, str3, str4;

bool end;

EdgeIterator(Graph<VertexT, EdgeT> &g) {

graph = &g;

if(graph->Dense()) {

mIt = new GraphMatrixForm<EdgeT>::EdgeIterator(static\_cast<GraphMatrixForm<EdgeT>\*>(g.data));

useM = true;

} else {

lIt = new GraphListForm<EdgeT>::EdgeIterator(static\_cast<GraphListForm<EdgeT>\*>(g.data));

useM = false;

}

//begin();

}

//Установка в начало

bool begin() {

if(useM) return

mIt->begin();

else

return lIt->begin();

}

//Сдвиг на следующий элемент

bool operator++() {

if(useM)

return mIt->next();

else

return lIt->next();

}

//Проверка на выход

bool onEnd() {

if(useM)

return mIt->onEnd();

else

return lIt->onEnd();

}

//Установить итератор рёбер на конечное ребро

bool toend(){

if (useM)

return mIt->toend();

else

return lIt->toend();

}

//Получение ребра

EdgeT\* operator\*() {

if (onEnd())

throw "Exeption#";

if(useM)

return mIt->getEdge();

else

return lIt->getEdge();

}

//Прочитать вершины ребра и вес по текущей позиции итератора

string read\_edge(){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\*){

throw "Exeption#";

}

VertexT \*\_v1, \*\_v2;

\_v1=e->getVertex1();

\_v2=e->getVertex2();

if (\_v1->nameSet==false)

str1="#data not available";

else

str1=\_v1->getName();

if (\_v2->nameSet==false)

str2="#data not available";

else

str2=\_v2->getName();

if (e->isWeightSet()==false)

str3="#data not available";

else

str3=i\_to\_s(e->getWeight());

str4="Исходящая вершина: "+str1+". Входящая вершина: "+str2+". Вес ребра: "+str3;

return str4;

}

//Прочитать данные ребра по текущей позиции итератора

int read\_data\_edge(){

EdgeT\* f;

try{

f=operator\*();

}

catch(char\*e){

throw "Exeption#";

}

if (f->isDataSet())

return f->getData();

else

throw "#data not available";

};

bool write\_data\_edge(int d){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\*e){

return false;

}

e->setData(d);

return true;

};

bool write\_weight\_edge(int w){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

return false;

}

e->setWeight(w);

return true;

}

};

//Итератор исходящих рёбер

class OutputEdgeIterator {

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

typename GraphMatrixForm<EdgeT>::OutputEdgeIterator \*mIt;

typename GraphListForm<EdgeT>::OutputEdgeIterator \*lIt;

bool useM;

public:

string str1, str2, str3, str4;

int curV1, curV2;

bool end;

OutputEdgeIterator(Graph<VertexT, EdgeT> &g, VertexT &v) {

graph = &g;

curV1 = graph->getIndex(&v);

if(graph->Dense()) {

mIt = new GraphMatrixForm<EdgeT>::OutputEdgeIterator(static\_cast<GraphMatrixForm<EdgeT>\*>(g.data), curV1);

useM = true;

} else {

lIt = new GraphListForm<EdgeT>::OutputEdgeIterator(static\_cast<GraphListForm<EdgeT>\*>(g.data), curV1);

useM = false;

}

}

//Установка в начало

bool begin() {

if(useM)

return mIt->begin();

else

return lIt->begin();

}

//Сдвиг на следующий элемент

bool operator++() {

if(useM)

return mIt->next();

else

return lIt->next();

}

bool toend(){

if(useM)

return mIt->toend();

else

return lIt->toend();

}

//Проверка на выход

bool onEnd() {

if(useM)

return mIt->onEnd();

else

return lIt->onEnd();

}

//Получение ребра

EdgeT\* operator\*() {

if (onEnd())

throw "#Exeption";

if(useM)

return mIt->getEdge();

else

return lIt->getEdge();

}

string read\_edge(){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\*f){

throw "Exeption#";

}

VertexT \*\_v1, \*\_v2;

\_v1=e->getVertex1();

\_v2=e->getVertex2();

if (\_v1->nameSet==false)

str1="#data not available";

else

str1=\_v1->getName();

if (\_v2->nameSet==false)

str2="#data not available";

else

str2=\_v2->getName();

if (e->isWeightSet()==false)

str3="#data not available";

else

str3=i\_to\_s(e->getWeight());

str4="Исходящая вершина: "+str1+". Входящая вершина: "+str2+". Вес ребра: "+str3;

return str4;

}

int read\_data\_edge(){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* f){

throw "Exeption#";

}

int a;

if (e->isDataSet())

return e->getData();

else

throw "#data not available";

};

bool write\_data\_edge(int d){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

return false;

}

e->setData(d);

return true;

};

bool write\_weight\_edge(int w){

EdgeT\* e;

try{

e=operator\*();

}

catch(char\* Error){

return false;

}

e->setWeight(w);

return true;

}

};

//двухпроходной эйлеров цикл

class Euler {

int \*color; //цвет вершины 0-белый, 1-серый, 2-черный

int \*p; //подграф предшествования

int \*\*Adj; //матрица смежных вершин

int n; //кол верш в графе

stack <int> st;

Graph<VertexT, EdgeT> \*graph; //Обрабатываемый граф

public:

typename Graph<VertexT, EdgeT>::OutputEdgeIterator \*out;

Euler(Graph<VertexT, EdgeT> &g) {

graph = &g;

}

Euler(const Graph<VertexT, EdgeT> &T) {

graph = T.graph;

}

~Euler() {

delete color,p,Adj;

}

bool DFS () {

if (graph->directed) {

return false;

}

n=graph->V();

color=new int[n];

p=new int [n];

Adj = new int \*[n];

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

Adj[i]=new int [n];

}

for (int u=0; u<n; u++) {

color[u]=0;

p[u]=-1;

}

search ();

for (int u=0; u<n; u++) {

if (color[u]==0) {

DFS\_Visit(u);

}

}

show();

return true;

}

void DFS\_Visit (int u) {

color[u]=1;

//------------------------------------------------

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

if (Adj[u][i]==INT\_MAX) {

break;

}

if (color[Adj[u][i]]==0) {

st.push(u);

st.push(Adj[u][i]);

//------------------------------------------------

p[Adj[u][i]]=u;

DFS\_Visit(Adj[u][i]);

//------------------------------------------------

st.push(Adj[u][i]);////

st.push(u);////

//------------------------------------------------

}

else if (color[Adj[u][i]]==1) {

if (p[u]!=Adj[u][i]) {

st.push(u);

st.push(Adj[u][i]);

st.push(Adj[u][i]);

st.push(u);

}

}

color[u]=2;

}

}

void search () {

int vv1,vv2;

for (int u=0; u<n; u++) {

try {

out=new Graph<VertexT, EdgeT>::OutputEdgeIterator(\*graph,\*graph->getVertex(u));

}

catch (char \*) {

continue;

}

if (!out->begin()) {

continue;

}

for (int j=0; j<n; j++) {

try {

out->read\_edge();

}

catch (char \*) {

Adj[u][j]=INT\_MAX;

break;

}

stringstream stream1(out->str1);

stream1>>vv1;

stringstream stream2(out->str2);

stream2>>vv2;

if (u==vv2) {

Adj[u][j]=vv1;

}

else {

Adj[u][j]=vv2;

}

try {

++\*out;

}

catch (char \*) {

break;

}

}

}

}

void show () {

int \*temp;

temp=new int [st.size()];

int st\_size=st.size();

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

temp[i]=st.top();

st.pop();

}

//-----------------------------------------------------

//вывод попарно

for (int i=(st\_size-1); i>-1; i--) {

cout<<temp[i]<<" ";

}

//-----------------------------------------------------

delete temp;

}

};

//алгоритм Беллмана - Форда

class Bellman\_ford {

int \*\*mtr;

struct edges {

int a, b, cost;

} edg;

int n, m, v;

vector<edges> e;

int INF;

Graph<VertexT, EdgeT> \*graph;

public:

typename Graph<VertexT, EdgeT>::OutputEdgeIterator \*out;

Bellman\_ford(Graph<VertexT, EdgeT> &g) {

graph = &g;

INF = 1000000000;

}

Bellman\_ford(const Graph<VertexT, EdgeT> &T) {

graph = T.graph;

}

~Bellman\_ford () {

delete mtr;

}

void search () {

int sz=graph->V();

int vv1,vv2,w;

for (int u=0; u<sz; u++) {

try {

out=new Graph<VertexT, EdgeT>::OutputEdgeIterator(\*graph,\*graph->getVertex(u));

}

catch (char \*) {

continue;

}

if (!out->begin()) {

continue;

}

for (int j=0; j<sz; j++) {

try {

out->read\_edge();

}

catch (char \*) {

break;

}

stringstream stream1(out->str1);

stream1>>vv1;

stringstream stream2(out->str2);

stream2>>vv2;

stringstream stream3(out->str3);

stream3>>w;

if (u==vv2) {

vv2=vv1;

}

edg.a=u; //исходящая

edg.b=vv2; //входящая

edg.cost=w; //вес

e.push\_back(edg);

try {

++\*out;

}

catch (char \*) {

break;

}

}

}

}

string bellman\_ford() {

search();

n=m=graph->E();

mtr = new int \*[n];

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

mtr[i]=new int [n];

for (int j=0; j<n; j++) {

mtr[i][j]=INF;

}

}

for (int v=0; v<n; v++) {

vector<int> d (n, INF);

d[v] = 0;

vector<int> p (n, -1);

for (;;) {

bool any = false;

for (int j=0; j<m; ++j) {

if (d[e[j].a] < INF) {

if (d[e[j].b] > d[e[j].a] + e[j].cost) {

d[e[j].b] = d[e[j].a] + e[j].cost;

p[e[j].b] = e[j].a;

any = true;

}

}

}

//-----------------------------------------------------

int x;

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

x = -1;

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

if (d[e[j].a] < INF)

if (d[e[j].b] > d[e[j].a] + e[j].cost) {

d[e[j].b] = max (-INF, d[e[j].a] + e[j].cost);

p[e[j].b] = e[j].a;

x = e[j].b;

}

}

if (x!=-1) {

return "отрицательный цикл";

}

//-----------------------------------------------------

if (!any) break;

}

for (int h=0; h<n; h++) {

mtr[v][h]=d[h];

}

}

return centr();

}

string centr () {

int size=graph->V();

int \*temp; //массив экстрентис.

temp = new int [size];

bool wr;

for (int k=0; k<size; k++) {

temp[k]=INF;

}

//поиск максимальных элементов

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

wr=false;

for (int j=0; j<size; j++) {

if ((mtr[i][j]==INF) || mtr[i][j]==0) {

continue;

}

if (!wr) {

temp[i]=mtr[i][j];

wr=true;

}

else {

if (temp[i]<mtr[i][j]) {

temp[i]=mtr[i][j];

}

}

}

}

//определение центра

int result=0;

int tmp=0;

tmp=temp[0];

for (int u=1; u<size; u++) {

if (tmp>temp[u]) {

tmp=temp[u];

result=u;

}

}

if (tmp==INF || tmp==0) {

return "Центр не найден";

}

//-------------------------------------------------------------------------

//показать матрицу

cout<<endl<<"=============================================="<<endl<<" ";

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

cout<<i<<" ";

}

cout<<endl<<"-----------------------------------------------"<<endl;

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

cout<<i<<" | ";

for (int y=0; y<size; y++) {

if (mtr[i][y]==INF) {

cout<<"-"<<" ";

}

else {

cout<<mtr[i][y]<<" ";

}

}

if ((temp[i]==INF) || (temp[i]==0)) {

cout<<"| -"<<endl;

continue;

}

else {

cout<<"| "<<temp[i]<<endl;

}

}

delete temp;

//-----------------------------------------------------------------------

stringstream str;

string result0;

str << result;

str >> result0;

return "Цент графа, узел: "+result0;

}

};

};

Main.cpp

#include "Graph.h"

void main() {

int menu;

bool on=false; //Итератор исход реб

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>> \*graph;

Vertex<string, int> \*vertex;

Edge<Vertex<string, int>, int, int> \*edge;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::VertexIterator \*vIt;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::EdgeIterator \*eIt;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::OutputEdgeIterator \*oIt;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::Euler \*eIl;

Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::Bellman\_ford \*BF;

setlocale (0, "Rus");

cout<<"1. Создать пустой L-граф с нулевым числом вершин и ребер"<<endl;

cout<<"2. Создать граф с V вершинами, без ребер, типа D, формы представления F"<<endl;

cout<<"3. Создать граф с V вершинами, с E случайными ребрами, типа D , формы F"<<endl;

cout<<"4. Выход"<<endl;

while (true){

cout<<"Выберите операцию: ";

cin>>menu;

switch (menu){

case 1:{ //Создать пустой L-граф с нулевым числом вершин и ребер

graph=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>();

vIt=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::VertexIterator(\*graph);

eIt = new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::EdgeIterator(\*graph);

cout<<endl<<"Граф создан"<<endl;

break;

};

case 2:{ //Создать граф с V вершинами, без ребер, типа D, формы представления F

int tmp1,tmp2,tmp3;

cout<<"Введите количество вершин: ";

cin>>tmp1;

cout<<"Ориентированный ли граф (0 - нет, 1 - да): ";

cin>>tmp2;

if (tmp2==1) on=true;

cout<<"Форму графа (0 - список, 1 - матрица): ";

cin>>tmp3;

graph=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>(tmp1, tmp2, tmp3);

vIt=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::VertexIterator(\*graph);

eIt = new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::EdgeIterator(\*graph);

cout<<endl<<"Граф создан"<<endl;

break;

};

case 3:{ //Создать граф с V вершинами, с E случайными ребрами, типа D , формы F

int tmp1,tmp2,tmp3,tmp4;

cout<<"Введите количество вершин: ";

cin>>tmp1;

cout<<"Введите количество ребер: ";

cin>>tmp2;

cout<<"Ориентированный ли граф (0 - нет, 1 - да): ";

cin>>tmp3;

if (tmp3==1) on=true;

cout<<"Форму графа (0 - список, 1 - матрица): ";

cin>>tmp4;

graph=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>(tmp1, tmp2, tmp3, tmp4);

vIt=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::VertexIterator(\*graph);

eIt = new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::EdgeIterator(\*graph);

cout<<endl<<"Граф создан"<<endl;

break;

};

case 4: { //выход

return;

};

default: {

cout<<"Ошибка ввода"<<endl;

};

}

break;

}

while (true){

cout<<"0. Распечатать граф"<<endl;

cout<<"1. Опрос числа вершин в графе"<<endl;

cout<<"2. Опрос числа ребер в графе"<<endl;

cout<<"3. Опрос типа графа"<<endl;

cout<<"4. Опрос формы представления графа"<<endl;

cout<<"5. Опрос коэффициента насыщенности графа"<<endl;

cout<<"6. Преобразовать граф к L-форме"<<endl;

cout<<"7. Преобразовать граф к M-форме"<<endl;

cout<<"8. Добавить вершину к графу"<<endl;

cout<<"9. Удалить вершину из графа"<<endl;

cout<<"10. Вставить ребро в граф"<<endl; //тут вообще фигня

cout<<"11. Удалить ребро из графа"<<endl;

cout<<endl<<"Операции над вершинами:"<<endl;

cout<<"12. Прочитать данные о вершине"<<endl;

cout<<"13. Записать данные о вершине"<<endl;

cout<<"14. Перезаписать имя вершины"<<endl;

cout<<endl<<"Операции над ребрами:"<<endl;

cout<<"15. Прочитать вес ребра"<<endl;

cout<<"16. Прочитать данные о ребре"<<endl;

cout<<"17. Записать данные о ребре"<<endl;

cout<<"18. Записать вес ребра"<<endl;

cout<<endl<<"Итератор вершин:"<<endl;

cout<<"19. Установить итератор вершин на начальную вершину"<<endl;

cout<<"20. Установить итератор вершин на конечную вершину"<<endl;

cout<<"21. Переход к следующей позиции"<<endl;

cout<<"22. Прочитать вершину по текущей позиции итератора"<<endl;

cout<<"23. Прочитать данные вершины по текущей позиции итератора"<<endl;

cout<<"24. Записать данные вершины по текущей позиции итератора"<<endl;

cout<<"25. Записать имя вершины по текущей позиции итератора"<<endl;

cout<<endl<<"Итератор ребер:"<<endl;

cout<<"26. Установить итератор ребер на начальное ребро"<<endl;

cout<<"27. Установить итератор ребер на конечное ребро"<<endl;

cout<<"28. Переход к следующей позиции"<<endl;

cout<<"29. Прочитать вершины ребра и вес по текущей позиции итератора"<<endl;

cout<<"30. Прочитать данные ребра по текущей позиции итератора"<<endl;

cout<<"31. Записать данные о ребре по текущей позиции итератора"<<endl;

cout<<"32. Записать вес ребра по текущей позиции итератора"<<endl;

cout<<endl<<"33. Итератор исходящих ребер"<<endl;

cout<<endl<<"35. Двухпроходной Эйлеров цикл"<<endl;

cout<<endl<<"36. Алгоритм Беллмана - Форда"<<endl;

cout<<endl<<"40. Выход"<<endl;

cout<<endl<<"Выберите операцию: ";

cin>>menu;

switch (menu){

case 0:{

graph->print\_graph();

break;

};

case 1:{

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Число вершин в графе: "<<graph->V()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 2:{

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Число ребер в графе: "<<graph->E()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 3:{

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Тип графа (0 - неориентированный, 1 - ориентированный): "<<graph->Directed()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 4:{

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Форма предстваления графа (0 - L-граф, 1 - M-граф): "<<graph->Dense()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 5:{

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Коэффициент насыщенности: "<<graph->K()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 6:{

graph->ToListGraph();

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Граф преобразован"<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 7:{

graph->ToMatrixGraph();

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Граф преобразован"<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 8:{

cout<<endl<<"=============================================="<<endl;

if(graph->InsertV()) cout<<endl<<"Вершина добавлена в граф"<<endl;

else cout<<endl<<"Вершина не добавлена в граф"<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 9:{

string v1;

cout<<"Введите удаляемую вершину: ";

cin>>v1;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Вершина удалена (1 - да, 0 - нет): "<<graph->DeleteV(v1)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

}

case 10:{

string v1, v2;

bool tmp;

cout<<"Введите вершину, из которой выходит ребро: ";

cin>>v1;

cout<<"Введите вершину, в которую входит ребро: ";

cin>>v2;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Ребро добавлено в граф (1 - да, 0 - нет): "<<graph->InsertE(v1, v2)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 11:{

string v1, v2;

cout<<"Введите вершину для удаления 1: ";

cin>>v1;

cout<<"Введите вершину для удаления 2: ";

cin>>v2;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Ребро удалено (1 - да, 0 - нет): "<<graph->DeleteE(v1, v2)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

//-------------------------------------------------------

//Операции над вершинами

case 12:{ //Прочитать данные о вершине

int tmp;

string v;

cout<<"Введите имя вершины: ";

cin>>v;

cout<<endl<<"=============================================="<<endl;

tmp=graph->read\_data\_vertex(v);

if (tmp!=-1) cout<<endl<<"Данные вершины: "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 13:{ //Записать данные о вершине

int tmp;

string v;

cout<<"Введите имя вершины: ";

cin>>v;

cout<<"Введите данные: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Данные о вершине записаны (1 - да, 0 - нет): "<<graph->write\_data\_vertex(v, tmp)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 14:{ //Записать новое имя вершины

string v1, v2;

cout<<"Введите старое имя вершины: ";

cin>>v1;

cout<<"Введите новое имя: ";

cin>>v2;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Имя вершины перезаписано (1 - да, 0 - нет): "<<graph->write\_name\_vertex(v1, v2)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

//-------------------------------------------------------

//Операции над ребрами

case 15:{ //Прочитать вес ребра

int tmp;

string v1, v2;

cout<<"Введите имя исходящей вершины: ";

cin>>v1;

cout<<"Введите имя входящей вершины: ";

cin>>v2;

cout<<endl<<"=============================================="<<endl;

tmp=graph->read\_weight\_edge(v1, v2);

if (tmp!=-1) cout<<endl<<"Вес ребра: "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 16:{ //Прочитать данные о ребре

int tmp;

string v1, v2;

cout<<"Введите имя исходящей вершины: ";

cin>>v1;

cout<<"Введите имя входящей вершины: ";

cin>>v2;

cout<<endl<<"=============================================="<<endl;

tmp=graph->read\_data\_edge(v1, v2);

if (tmp!=-1) cout<<endl<<"Данные ребра: "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 17:{ //Записать данные о ребре

int tmp;

bool tmp1;

string v1, v2;

cout<<"Введите имя исходящей вершины: ";

cin>>v1;

cout<<"Введите имя входящей вершины: ";

cin>>v2;

cout<<"Введите данные: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

tmp1=graph->write\_data\_edge(v1, v2, tmp);

if (tmp1==true) cout<<endl<<"Данные о вершине записаны (1 - да, 0 - нет): "<<tmp1<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 18:{ //Записать вес ребра

int tmp;

bool tmp1;

string v1, v2;

cout<<"Введите имя исходящей вершины: ";

cin>>v1;

cout<<"Введите имя входящей вершины: ";

cin>>v2;

cout<<"Введите вес: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

tmp1=graph->write\_weight\_edge(v1, v2, tmp);

if (tmp1==true) cout<<endl<<"Вес ребра перезаписан (1 - да, 0 - нет): "<<tmp1<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

//-------------------------------------------------------

//Итератор вершин

case 19:{ //Установить итератор вершин на начальную вершину

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор вершин графа установлен на первую вершину (1 - да, 0 - нет): "<<vIt->begin()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 20:{ //Установить итератор вершин на конечную вершину

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор вершин графа установлен на последнюю вершину (1 - да, 0 - нет): "<<vIt->toend()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 21:{ //Переход к следующей позиции

bool tmp;

cout<<endl<<"=============================================="<<endl;

tmp=++\*vIt;

if (tmp) cout<<endl<<"Итератор вершин графа установлен на следующую позицию (1 - да, 0 - нет): "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 22:{ //Прочитать вершину по текущей позиции итератора

string tmp;

cout<<endl<<"=============================================="<<endl;

try {

tmp=vIt->read\_vertex();

}

catch (char\*e) {

cerr<<endl<<e<<endl;

cout<<endl<<"=============================================="<<endl;

break;

}

cout<<endl<<"Вершина на текущей позиции итератора: "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 23:{ //Прочитать данные вершины по текущей позиции итератора

int tmp;

cout<<endl<<"=============================================="<<endl;

tmp=vIt->read\_data\_vertex();

if (tmp!=-1) cout<<endl<<"Данные вершины на текущей позиции итератора: "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 24:{ //Записать данные о вершины по текущей позиции итератора

int tmp;

bool tmp2;

cout<<"Введите данные: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

tmp2=vIt->write\_data\_vertex(tmp);

if (tmp2) cout<<endl<<"Данные вершины записаны по текущей позиции итератора (1 - да): "<<tmp2<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 25:{ //Записать имя вершины по текущей позиции итератора

string cur;

bool tmp;

cout<<"Введите имя: ";

cin>>cur;

cout<<endl<<"=============================================="<<endl;

tmp=vIt->write\_name\_vertex(cur);

if (tmp) cout<<endl<<"Имя вершины записано по текущей позиции итератора (1 - да): "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

//-------------------------------------------------------

//Итератор ребер

case 26:{ //Установить итератор ребер на начальное ребро

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор установлен на начальное ребро (1 - д, 0 - нет): "<<eIt->begin()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 27:{ //Установить итератор ребер на конечное ребро

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор установлен на последнее ребро (1 - да, 0 - нет): "<<eIt->toend()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 28:{ //Переход к следующей позиции

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор ребер графа установлен на следующую позицию (1 - да, 0 - нет): "<<++\*eIt<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 29:{ //Прочитать вершины ребра и вес по текущей позиции итератора

string tmp;

cout<<endl<<"=============================================="<<endl;

try {

tmp=eIt->read\_edge();

}

catch (char\*e) {

cerr<<endl<<e<<endl;

cout<<endl<<"=============================================="<<endl;

break;

}

cout<<endl<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 30:{ //Прочитать данные ребра по текущей позиции итератора

int tmp;

cout<<endl<<"=============================================="<<endl;

try{

tmp=eIt->read\_data\_edge();

}

catch (char\*e){

cerr<<endl<<e<<endl;

cout<<endl<<"=============================================="<<endl;

break;

}

cout<<endl<<"Данные ребра по текущей позиции итератора: "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 31:{ //Записать данные о ребре по текущей позиции итератора

int tmp;

cout<<"Введите данные: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Данные ребра записаны по текущей позиции итератора (1 - да, 0 - нет): "<<eIt->write\_data\_edge(tmp)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 32:{ //Записать вес ребра по текущей позиции итератора

int tmp;

cout<<"Введите вес: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Вес ребра записан по текущей позиции итератора (1 - да, 0 - нет): "<<eIt->write\_weight\_edge(tmp)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

//-------------------------------------------------------

//Итератор исходящих ребер

case 33: {

int tmp,tmp2; //узел

bool ex=false; //выход

cout<<endl<<"Введите веришну: ";

cin>>tmp;

oIt=new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::OutputEdgeIterator(\*graph, \*graph->getVertex(tmp));

while (true) {

cout<<endl<<"Итератор исходящих ребер:"<<endl;

cout<<"1. Установить итератор исходящих ребер на начальное ребро"<<endl;

cout<<"2. Установить итератор исходящих ребер на конечное ребро"<<endl;

cout<<"3. Переход к следующей позиции"<<endl;

cout<<"4. Прочитать вершины ребра и вес по текущей позиции итератора"<<endl;

cout<<"5. Прочитать данные ребра по текущей позиции итератора"<<endl;

cout<<"6. Записать данные о ребре по текущей позиции итератора"<<endl;

cout<<"7. Записать вес ребра по текущей позиции итератора"<<endl;

cout<<"10. Вернуться в предыдущее меню"<<endl;

cout<<endl<<"Выберите операцию: ";

cin>>tmp2;

switch (tmp2) {

case 1:{ //Установить итератор исходящих ребер на начальное ребро

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор исходящих ребер установлен на первое ребро (1 - да, 0 - нет): "<<oIt->begin()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 2:{ //Установить итератор исходящих ребер на конечное ребро

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор исходящих ребер установлен на последнее ребро (1 - да, 0 - нет): "<<oIt->toend()<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 3:{ //Переход к следующей позиции

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Итератор исходящих ребер установлен на следующую позицию (1 - да, 0 - нет): "<<++\*oIt<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 4:{ //Прочитать вершины ребра и вес по текущей позиции итератора

string tmp;

cout<<endl<<"=============================================="<<endl;

try{

tmp=oIt->read\_edge();

}

catch(char\*e){

cerr<<endl<<e<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

cout<<endl<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 5:{ //Прочитать данные ребра по текущей позиции итератора

int tmp;

cout<<endl<<"=============================================="<<endl;

try{

tmp=oIt->read\_data\_edge();

}

catch (char\*e){

cerr<<endl<<e<<endl;

cout<<endl<<"=============================================="<<endl;

break;

}

cout<<endl<<"Данные ребра по текущей позиции итератора: "<<tmp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 6:{ //Записать данные о ребре по текущей позиции итератора

int tmp;

cout<<"Введите данные: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Данные ребра записаны по текущей позиции итератора (1 - да, 0 - нет): "<<oIt->write\_data\_edge(tmp)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 7:{ //Записать вес ребра по текущей позиции итератора

int tmp;

cout<<"Введите вес: ";

cin>>tmp;

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Вес ребра записан по текущей позиции итератора (1 - да, 0 - нет): "<<oIt->write\_weight\_edge(tmp)<<endl;

cout<<endl<<"=============================================="<<endl;

break;

};

case 10: { //выход

ex=true;

break;

};

default: {

cout<<"Ошибка ввода"<<endl;

};

}

if (ex) break;

}

break;

}

case 35: { //формирование списка ребер неориентированного графа в порядке двухпроходного эйлерова цикла

eIl = new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::Euler(\*graph);

bool temp;

cout<<endl<<"=============================================="<<endl;

temp=eIl->DFS();

if (!temp) cout<<endl<<"Эйлеров цикл: "<<temp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

}

case 36: { //определение центра взвешенного орграфа на основе алгоритма Беллмана - Форда

BF = new Graph<Vertex<string, int>, Edge<Vertex<string, int>, int, int>>::Bellman\_ford(\*graph);

string temp;

temp=BF->bellman\_ford();

cout<<endl<<"=============================================="<<endl;

cout<<endl<<"Беллман - Форд: "<<temp<<endl;

cout<<endl<<"=============================================="<<endl;

break;

}

case 40: { //выход

return;

};

break;

default: {

cout<<"Ошибка ввода"<<endl;

};

break;

}

}

getch();

return;

}