МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ

РОССИЙСКОЙ ФЕДЕРАЦИИ

НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

ФАКУЛЬТЕТ АВТОМАТИКИ И ВЫЧИСЛИТЕЛЬНОЙ ТЕХНИКИ

Кафедра Вычислительной Техники

Расчётно-графическая работа по дисциплине

«Алгоритмы и структуры данных»

на тему «**Простой граф**»

Группа: АВТ-610

Студент: Дунаев Н. Ю.

Варианты: 3, 10

Преподаватель: Романенко Т. А.

Новосибирск 2018

Оглавление

[1. Цель работы 3](#_Toc533195121)

[2. Общее задание 3](#_Toc533195122)

[3. Диаграмма взаимосвязи объектов, реализующих АТД «Простой граф», вспомогательных объектов, АТД задач 1 и 2 6](#_Toc533195123)

[4. Формат АТД “ Простой граф ” 7](#_Toc533195124)

[5. Клиентское определение класса «Простой граф» 9](#_Toc533195125)

[6. Формат АТД «Дескриптор вершины» 10](#_Toc533195126)

[7. Клиентское определение класса «Дескриптор вершины» 11](#_Toc533195127)

[8. Формат АТД «Дескриптор ребра» 12](#_Toc533195128)

[9. Клиентское определение класса «Дескриптор ребра» 13](#_Toc533195129)

[10. Формат АТД «Итератор вершин» 14](#_Toc533195130)

[11. Клиентское определение класса «Итератор вершин» 15](#_Toc533195131)

[12. Формат АТД «Итератор ребер» 15](#_Toc533195132)

[13. Клиентское определение класса «Итератор ребер» 16](#_Toc533195133)

[14. Формат АТД «Итератор исходящих ребер» 16](#_Toc533195134)

[15. Клиентское определение класса «Итератор исходящих ребер» 17](#_Toc533195135)

[16. Формат АТД «Задача 1» 18](#_Toc533195136)

[17. Клиентское определение класса «Задача 1» 18](#_Toc533195137)

[18. Формат АТД «Задача 2» 19](#_Toc533195138)

[19. Клиентское определение класса «Задача 2» 19](#_Toc533195139)

[20. Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 1 20](#_Toc533195140)

[21. Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 2 20](#_Toc533195141)

[Заключение 20](#_Toc533195142)

[Список использованных источников 21](#_Toc533195143)

[Приложение 22](#_Toc533195144)

# Цель работы

Освоение технологии разработки комплексного программного обеспечения для решения задач в различных прикладных областях.

# Общее задание

1. 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

**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. 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

**Вариант задачи 1:**

1. Определение гамильтонова цикла в орграфе.

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

Интерфейс ***АТД «Задача 2»*** 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

**Вариант задачи 2:**

1. Нахождения ребра (ребер), устранение которых вызывает максимальное возрастание кратчайшего пути из вершины u в вершину v во взвешенном орграфе на основе алгоритма Дейкстры.

# Диаграмма взаимосвязи объектов, реализующих АТД «Простой граф», вспомогательных объектов, АТД задач 1 и 2

**Вершина графа**

**template <class NameT,**

**class DataT>**

**class Vertex:**

NameT name – имя вершины

DataT data – данные вершины

**Ребро графа**

**template <class VertexT,**

**class WeighT,**

**class DataT>**

**Class Edge:**

VertexT v1, v2 – соединяемые вершины DataT data – данные WeightT weight - вес

**Матрица смежности**

**template <class EdgeT>**

**class GraphMatrixForm:**

**pablic GraphForm<EdgeT>**

vector<vector<EdgeT\*>>matrix – матрица смежности

**Список рёбер**

**template <class EdgeT>**

**Class GraphListForm:**

**Public GraphForm<EdgeT>**

Class Node {

public:

EdgeT \*edge – ребро

Int v2; - вторая вершина

};

Vector<list<Node>> edgeList;

- список рёбер

**Абстрактный граф**

**template <class EdgeT> class GraphForm**

**Класс графа**

**template <class VertexT, class EdgeT>**

**class Graph:**

vector<VertexT\*>vertex Vector;

GraphForm<EdgeT> \*data;

bool directed;

bool dense;

Методы работы с графом:

Итератор вершин -

**class VertexIterator<VertexT,EdgeT>**

Итератор ребер -

**class EdgeIterator<VertexT,EdgeT>**

Итератор исходящих рёбер -

**class OutputEdgeIterator<VertexT,EdgeT>**

**Задача 1**

**template<class VertexT,class EdgeT> class Task1;**

Graph<VertexT,EdgeT> \*graph;

**Задача 2**

**template<class VertexT,class EdgeT> class Task2;**

Graph<VertexT,EdgeT> \*graph;

# 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.

*Параметры:*

Коллекция вершин;

Коллекция рёбер;

Тип графа **dense**;

Ориентированность графа **directed**.

***Операции:***

*Конструктор без параметров*

**Вход:** нет

**Предусловие:** нет

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

**Выход:** нет

**Постусловия:** создан пустой граф

*Конструктор с параметрами*

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

**Предусловие:** нет

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

**Выход:** нет

**Постусловия:** создан граф с V вершинами

*Конструктор с параметрами*

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

**Предусловие:** 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

**Предусловие:** нет

**Процесс:** копирование данных графа G

**Выход:** нет

**Постусловия:** создана копия графа

*Деструктор*

**Вход:** нет

**Предусловия:** нет

**Процесс:** удаление коллекции

**Выход:** нет

**Постусловия:** коллекция удалена

*Получение числа вершин в графе*

**Вход:** нет

**Предусловие:** нет

**Процесс:** получение размера коллекции вершин графа

**Выход:** число вершин в графе

**Постусловия:** нет

*Получение числа вершин в графе*

**Вход:** нет

**Предусловие:** нет

**Процесс:** получение размера коллекции рёбер графа

**Выход:** число ребер в графе

**Постусловия:** нет

*Получение коэффициента заполнености графа*

**Вход:** нет

**Предусловие:** нет

**Процесс:** подсчёт коэффициента корреляции

**Выход:** коэффициент корреляции графа

**Постусловия:** нет

*Получение направленности графа*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка параметра directed

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

**Постусловия:** нет

*Получение формы представления графа*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка параметра dense

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

**Постусловия:** нет

*Преобразование графа к М-типу*

**Вход:** нет

**Предусловие:** нет

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

**Выход:** нет

**Постусловия:** граф принял M-форму

*Преобразование графа к L-типу*

**Вход:** нет

**Предусловие:** нет

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

**Выход:** нет

**Постусловия:** граф принял L-форму

*Вставка вершины в граф*

**Вход:** нет

**Предусловие:** 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

**Предусловие:** существование вершины V в графе

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

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

**Постусловия:** вершина V удалена из графа

*Получение вершины из графа*

**Вход:** имя вершины name

**Предусловия:** существование 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

***Конец АТД***

# 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 V(); //Получение числа вершин в графе

int E(); //Получение числа ребер в графе

bool Dense(); //Получение формы представления графа

bool Directed(); //Получение типа графа

double K(); //Получение коэффициента насыщенности графа

void ToListGraph(); //Преобразование граф ак L-типу

void ToMatrixGraph();//Преобразование графа к M-типу

VertexT\* InsertV(); //Вставка вершины в граф

bool DeleteV(VertexT \*v);//Удаление вершины из графа

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

bool DeleteE(VertexT \*v1, VertexT \*v2); //Удаление ребра из графа

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

bool hasEdge(VertexT \*v1, VertexT \*v2); //Проверка на существованиеие ребра в графе

};

# 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.

*Параметры:*

Имя вершины **name**;

Данные вершины **data**.

***Операции:***

*Конструктор без параметров*

**Вход:** нет

**Предусловие:** нет

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

**Выход:** нет

**Постусловия:** создана пустая вершина

*Конструктор с параметрами*

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

**Предусловие:** 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

**Предусловие:** нет

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

**Выход:** нет

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

*Установка данных вершины*

**Вход:** данные вершины data

**Предусловие:** нет

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

**Выход:** нет

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

*Получение данных вершины*

**Вход:** нет

**Предусловие:** имя было установлено

**Процесс:** обращение к полю объекта name

**Выход:** имя вершины name

**Постусловия:** нет

*Провекра установленности имени вершины*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка установленности поля name

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

**Постусловия:** нет

*Провекра установленности данных вершины*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка установленности поля data

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

**Постусловия:** нет

***Конец АТД***

# Клиентское определение класса «Дескриптор вершины»

template <class NameT, class DataT>

class Vertex

{

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

};

# 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.

*Параметры:*

Вес вершины **weight**;

Данные вершины **data**..

***Операции:***

*Конструктор с параметрами*

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

**Предусловие:** 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 с параметрами*

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

**Предусловие:** 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 с параметрами*

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

**Предусловие:** 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

**Предусловие:** нет

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

**Выход:** нет

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

*Установка данных ребра*

**Вход:** данные ребра data

**Предусловие:** нет

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

**Выход:** нет

**Постусловия:** данные ребра установлен в значение data

*Получение данных ребра*

**Вход:** нет

**Предусловие:** 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

**Предусловие:** вес был установлен

**Процесс:** обращение к полю объекта weight

**Выход:** вес вершины weight

**Постусловия:** нет

*Провекра установленности веса ребра*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка установленности поля weight

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

**Постусловия:** нет

*Провекра установленности данных ребра*

**Вход:** нет

**Предусловие:** нет

**Процесс:** проверка установленности поля data

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

**Постусловия:** нет

*Получение исходящей вершины ребра*

**Вход:** нет

**Предусловие:** нет

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

**Выход:** дескриптор исходящей вершины ребра

**Постусловия:** нет

*Получение входящей вершины ребра*

**Вход:** нет

**Предусловие:** нет

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

**Выход:** дескриптор входящей вершины ребра

**Постусловия:** нет

***Конец АТД***

# 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

};

# 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.

*Параметры:*

Текущая вершина в графе **current**;

Указатель на граф **graph**;

Признак конца просмотра **end**.

***Операции:***

*Конструктор с параметрами*

**Вход:** указатель на граф g

**Предусловие:** нет

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

**Выход:** нет

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

*Установка итератора на начало*

**Вход:** нет

**Предусловие:** 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

**Предусловие:** 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

**Предусловие:** нет

**Процесс:** проверка состояния итератора

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

**Постусловия:** нет

*Доступ к текущей вершине*

**Вход:** нет

**Предусловие:** итератор установлен

**Процесс:** доступ к текущей вершине

**Выход:** дескриптор вершины или исключение при невыполненном предусловии

**Постусловия:** нет

*Установка итератора на конец*

**Вход:** нет

**Предусловие:** нет

**Процесс:** установка итератора на последнюю вершину

**Выход:** нет

**Постусловия:** нет

***Конец АТД***

# Клиентское определение класса «Итератор вершин»

class VertexIterator

{

public:

VertexIterator(Graph<VertexT, EdgeT> &g){graph = &g;}; //Конструктор

bool begin();//Возвращает итератор, установленный на первую вершину графа

bool operator++(); //Переход к следующей позиции

bool toend();//Установить итератор вершин на конечную вершину

bool onEnd();//Проверка на выход

VertexT\* operator\*(); //Получение вершины

string read\_vertex();//Прочитать вершину по текущей позиции итератора

int read\_data\_vertex();//Прочитать данные вершины по текущей позиции итератора

bool write\_data\_vertex(int d); //Записать данные о вершины по текущей позиции итератора

bool write\_name\_vertex(string str) //Записать имя вершины по текущей позиции итератора

};

# 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.

*Параметры:*

Текущее ребро графа;

Указатель на граф **graph**

***Операции:***

*Конструктор с параметрами*

**Вход:** указатель на граф g

**Предусловие:** нет

**Процесс:** создание итератора ребер для графа G

**Выход:** нет

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

*Установка итератора на начало*

**Вход:** нет

**Предусловие:** 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

**Предусловие:** 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

**Предусловие:** нет

**Процесс:** проверка состояния итератора

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

**Постусловия:** нет

*Доступ к текущему ребру*

**Вход:** нет

**Предусловие:** итератор установлен

**Процесс:** доступ к текущему ребру

**Выход:** дескриптор ребра или исключение при невыполненном предусловии

**Постусловия:** нет

*Установка итератора на конец*

**Вход:** нет

**Предусловие:** проверка наличия рёбер

**Процесс:** установка итератора на последнее ребро

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

**Постусловия:** нет

***Конец АТД***

# Клиентское определение класса «Итератор ребер»

class EdgeIterator {

public:

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

bool begin();//Установка в начало

bool operator++();//Сдвиг на следующий элемент

bool onEnd();//Проверка на выход

bool toend();//Установить итератор рёбер на конечное ребро

EdgeT\* operator\*()//Получение ребра;

string read\_edge();//Прочитать вершины ребра и вес по текущей позиции итератора

int read\_data\_edge();//Прочитать данные ребра по текущей позиции итератора

bool write\_data\_edge(int d);//Записать данные ребра по текущей позиции итератора

bool write\_weight\_edge(int w); //Записать вес по текущей позиции итератора

};

# 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 последнее исходящее ребро вершины графа в порядке обхода, получать доступ к текущему ребру вершины графа, а также переходить к следующему.

*Параметры:*

Текущее ребро графа.

Указатель на граф **graph**.

Номер рассматриваемой вершины **curV1**.

***Операции:***

*Конструктор с параметрами*

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

**Предусловие:** 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 невыполнении предусловия

*Установка итератора на начало*

**Вход:** нет

**Предусловие:** 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

**Предусловие:** 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

**Предусловие:** нет

**Процесс:** проверка состояния итератора

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

**Постусловия:** нет

*Доступ к текущему ребру*

**Вход:** нет

**Предусловие:** итератор установлен

**Процесс:** доступ к текущему ребру

**Выход:** дескриптор исходящего ребра или исключение при невыполненном предусловии

**Постусловия:** нет

*Установка итератора на конец*

**Вход:** нет

**Предусловие:** проверка наличия исходящих рёбер

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

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

**Постусловия:** нет

***Конец АТД***

# Клиентское определение класса «Итератор исходящих ребер»

class OutputEdgeIterator {

public:

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

~OutputEdgeIterator(); //Деструктор

bool begin();//Установка в начало

bool operator++();//Сдвиг на следующий элемент

bool toend;//установка на последнее исходящее ребро

bool onEnd();//Проверка на выход

EdgeT\* operator\*();//Получение ребра

string read\_edge();();//Прочитать ребро по текущей позиции итератора

int read\_data\_edge();//Прочитать данные ребра по текущей позиции итератора

bool write\_data\_edge(int d);//Записать данные о ребре по текущей позиции итератора

bool write\_weight\_edge(int w);//Записать имя ребра по текущей позиции итератора

};

# Формат АТД «Задача 1»

Определение гамильтонова цикла в орграфе.

*Параметры:*

Обрабатываемый граф **graph**.

***Операции:***

*Конструктор с параметрами*

**Вход:** граф для решения задачи

**Предусловие:** нет

**Процесс:** привязка задачи к графу

**Выход:** нет

**Постусловия:** построен граф

*Конструктор копирования*

**Вход:** объект задачи

**Предусловие:** нет

**Процесс:** создание копии объекта задачи

**Выход:** нет

**Постусловия:** создана копия объекта задачи

*Получение решения*

**Вход:** орграф для решения задачи, стартовая вершина орграфа

**Предусловие:** нет

**Процесс:** определение гамильтонова цикла

**Выход:** нет

**Постусловия:** нет

*Повторное решение*

**Вход:** нет

**Предусловие:** нет

**Процесс:** определение гамильтонова цикла

**Выход:** нет

**Постусловия:** нет

*Вывод результата*

**Вход:** нет

**Предусловие:** нет

**Процесс:** нет

**Выход:** дек, содержащий последовательность вершин

**Постусловия:** нет

***Конец АТД***

# Клиентское определение класса «Задача 1»

template <class VertexT, class EdgeT>

class Task1

{

public:

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

Task1(Graph< VertexT, EdgeT> \*graph); // Конструктор с параметрами

Task1(Task1 \*obj); // Конструктор копирования

void set(Graph<VertexT, EdgeT> \*g, string v); // Получение решения

void restart(); // Повторное решение

deque<string> result(); // Вывод результата

};

# Формат АТД «Задача 2»

Нахождения ребра, устранение которого вызывает максимальное возрастание кратчайшего пути из вершины u в вершину v во взвешенном орграфе на основе алгоритма Дейкстры.

*Параметры:*

Обрабатываемый граф **graph**.

***Операции:***

*Конструктор с параметрами*

**Вход:** граф для решения задачи

**Предусловие:** нет

**Процесс:** привязка задачи к графу

**Выход:** нет

**Постусловия:** построен граф

*Конструктор копирования*

**Вход:** объект задачи

**Предусловие:** нет

**Процесс:** создание копии объекта задачи

**Выход:** нет

**Постусловия:** создана копия объекта задачи

*Получение решения*

**Вход:** исходящая вершина **v1,** входящая вершина **v2**

**Предусловие:** нет

**Процесс:** решение задачи

**Выход:** нет

**Постусловия:** нет

*Повторное решение*

**Вход:** исходящая вершина **v1,** входящая вершина **v2**

**Предусловие:** нет

**Процесс:** решение задачи

**Выход:** нет

**Постусловия:** нет

*Вывод результата*

**Вход:** нет

**Предусловие:** нет

**Процесс:** нет

**Выход:** массив издвух вершин, соединяемые устраняемым ребром

**Постусловия:** нет

***Конец АТД***

# Клиентское определение класса «Задача 2»

template <class VertexT, class EdgeT>

class Task2

public:

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

Task2(Graph< VertexT, EdgeT> \*graph); // Конструктор с параметрами

Task2(Task2 \*obj); // Конструктор копирования

void set(int v1, int v2); // Получение решения

void restart(); // Повторное решение

int\* result(); // Вывод результата

}

# Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 1

Алгоритм начинает выстраивать цикл от заданной исходной вершины при обходе графа в глубину. Очередному рекурсивному вызову DFS\_Visit задается дополнительный параметр - расчетная длина оставшегося пути до исходной вершины. При возврате из рекурсии возвращается признак удачного или неудачного построения пути заданной длины. Если возвращен признак удачного построения пути, то полученное решение включается в общее решение. Иначе неудачное решение отбрасывается вместе с пометкой вершины (цвет вершины становится белым). Это приводит к тому, что при новых попытках поиска другого пути эта вершина может быть выбрана вновь и будут исследованы заново все пути, исходящие из этой вершины.

Данная схема решения соответствует полному перебору всех перестановок ребер в графе и в пределе имеет трудоемкость **O(n!)**, то есть экспоненциальную трудоемкость.

# Краткое описание алгоритма, теоретическая оценка трудоемкости задачи 2

Задача 2 была реализованна с помощью алгоритма Дейкстры. Алгоритм Дейкстры перебирает все ребра и имеет трудоемкость **.** Алгоритм вызывается для каждого ребра на пути от начальной до конечной вершины, значит итоговая трудоемкость будет равна **.**

# 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.

Спроектирован и реализован шаблонный класс для АТД «Задача 1» для решения задачи: определение гамильтонова цикла в орграфе.

Спроектирован и реализован шаблонный класс для АТД «Задача 2» для решения задачи на взвешенном графе: нахождения ребра (ребер), устранение которых вызывает максимальное возрастание кратчайшего пути из вершины u в вершину v во взвешенном орграфе на основе алгоритма Дейкстры.

# Список использованных источников

1. Т. Кормен, Ч. Лейзерсон, Р. Ривест Алгоритмы. Анализ и построение. - М: БИНОМ, 2000 г. – 960 с.
2. Роберт Сэджвик. Фундаментальные алгоритмы на С++. Часть 5. Алгоритмы на графах - М: DiaSoft, 2002 г. – 496 с.
3. Альфред Ахо, Джон Э. Хопкрофт, Д. Ульман Структуры данных и алгоритмы. - М. - СПб – Киев: Вильямс, 2000 г. – 384 с.
4. Коллинз Уильям Дж. Структуры данных и стандартная библиотека шаблонов. – М.: ООО «Бином – Пресс», 2004 г. – 624 с.

# Приложение

Vertex.h

#pragma once

template <class NameT, class DataT>

class Vertex

{

NameT name; //Имя вершины

DataT data; //Данные вершины

bool nameIsSet, dataIsSet; //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

};

//Конструкторы

template <class NameT, class DataT>

Vertex <NameT, DataT>::Vertex() : nameIsSet(false), dataIsSet(false) {}

template <class NameT, class DataT>

Vertex <NameT, DataT>::Vertex(NameT name, DataT data) :

name(name),

data(data),

nameIsSet(true),

dataIsSet(true) {}

//Установка, получение имени и данных

template <class NameT, class DataT>

void Vertex <NameT, DataT>::setName(NameT name) {

this->name = name;

nameIsSet = true;

}

template <class NameT, class DataT>

void Vertex <NameT, DataT>::setData(DataT data) {

this->data = data;

dataIsSet = true;

}

template <class NameT, class DataT>

NameT Vertex <NameT, DataT>::getName() {

return name;

}

template <class NameT, class DataT>

DataT Vertex <NameT, DataT>::getData() {

return data;

}

template <class NameT, class DataT>

bool Vertex <NameT, DataT>::isNameSet() {

return nameIsSet;

}

template <class NameT, class DataT>

bool Vertex <NameT, DataT>::isDataSet() {

return dataIsSet;

}

Edge.h

#pragma once

template <class VertexT, class WeightT, class DataT>

class Edge {

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

WeightT weight; //Вес

DataT data; //Данные ребра

bool weightIsSet, dataIsSet;//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

};

//Конструкторы

template <class VertexT, class WeightT, class DataT>

Edge<VertexT, WeightT, DataT>::Edge(VertexT \*v1, VertexT \*v2) :

v1(v1),

v2(v2),

weightIsSet(false),

weight(0),

dataIsSet(false) {}

template <class VertexT, class WeightT, class DataT>

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

v1(v1),

v2(v2),

weight(weight),

weightIsSet(true),

dataIsSet(false) {}

template <class VertexT, class WeightT, class DataT>

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

v1(v1),

v2(v2),

weight(weight),

data(data),

weightIsSet(true),

dataIsSet(true) {}

//Установка и получение веса и данных

template <class VertexT, class WeightT, class DataT>

void Edge<VertexT, WeightT, DataT>::setWeight(WeightT weight)

{

this->weight = weight;

weightIsSet = true;

}

template <class VertexT, class WeightT, class DataT>

void Edge<VertexT, WeightT, DataT>::setData(DataT data)

{

this->data = data;

dataIsSet = true;

}

template <class VertexT, class WeightT, class DataT>

WeightT Edge<VertexT, WeightT, DataT>::getWeight()

{

return weight;

}

template <class VertexT, class WeightT, class DataT>

DataT Edge<VertexT, WeightT, DataT>::getData()

{

return data;

}

template <class VertexT, class WeightT, class DataT>

VertexT\* Edge<VertexT, WeightT, DataT>::getVertex1()

{

return v1;

}

template <class VertexT, class WeightT, class DataT>

VertexT\* Edge<VertexT, WeightT, DataT>::getVertex2()

{

return v2;

}

template <class VertexT, class WeightT, class DataT>

bool Edge<VertexT, WeightT, DataT>::isWeightSet()

{

return weightIsSet;

}

template <class VertexT, class WeightT, class DataT>

bool Edge<VertexT, WeightT, DataT>::isDataSet()

{

return dataIsSet;

}

Form.h

#pragma once

#include <vector>

#include <list>

using namespace std;

typedef Vertex <string, int> TVertex;

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;

virtual void print(vector<TVertex\*> vertexVector) = 0;

};

//форма представления матричная

//протестирован итератор ребер и итератор исходящих вершин

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

{

friend class EdgeIterator;

bool directed;

bool vector\_is\_contain(vector<string>& vec, string str) {

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

if (vec[i] == str)

return true;

return false;

}

public:

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

void print(vector<TVertex\*> vertexVector) {

int n = matrix.size();

vector<string> names;

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

names.push\_back(vertexVector[i]->getName());

cout << " ";

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

cout << names[i] << " ";

cout << endl;

int j;

bool check;

int i = 0;

for (vector<vector<EdgeT\*>>::iterator it = matrix.begin(); it != matrix.end(); it++, i++)

{

cout << names[i] << " ";

for (vector<EdgeT\*>::iterator it2 = it->begin(); it2 != it->end(); it2++)

{

if ((\*it2) != NULL)

cout << "1" << " ";

else

cout << "0" << " ";

}

cout << endl;

}

cout << endl;

}

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 если неор граф

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) {}

void print(vector<TVertex\*> vertexVector) {

vector<string> names;

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

names.push\_back(vertexVector[i]->getName());

vector<list<Node>>::iterator iterator = edgeList.begin();

int i = 0;

for (vector<list<Node>>::iterator it = edgeList.begin(); it != edgeList.end(); it++, i++)

{

cout << names[i] << ": ";

for (list<Node>::iterator it2 = it->begin(); it2 != it->end(); it2++)

{

//cout << (\*it2).edge->getVertex2()->getName();

cout << names[(\*it2).v2] << " ";

}

cout << endl;

}

cout << endl;

}

//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)

{

//Стираем симметричное ребро

if (!directed)

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

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

{

edgeList[index].erase(k);

break;

}

delete (\*j).edge;

edgeList[i].erase(j);

++deleted;

break;

}

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, до этого неправильно работал. щас норм

bool toend() {

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

bool flag = false;

for (curI = f->edgeList.size() - 1; curI >= 0; curI--) {

if (f->edgeList[curI].size() != 0) {

it = f->edgeList[curI].end();

it--;

curJ = it;

flag = true;

end = false;

return true;

}

}

end = true;

return false;

};

//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;

}

};

};

Graph.h

#pragma once

#include <iostream>

#include <sstream>

#include <vector>

using namespace std;

string int\_to\_string(int x) //Преобразование числа в строку

{

stringstream ss;

ss << x;

return ss.str();

}

template <class VertexT, class EdgeT>

class Graph

{

private:

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

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

bool directed; //Направленный граф

bool dense; //М-граф

int edgeCounter; //Число рёбер

int getIndexFromName(string n);//Получение индекса вершины по дескриптору

int sz;

bool vertex\_has\_name(string name);

public:

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 V();//возвращает число вершин в графе

int E(); //возвращает число ребер в графе

bool Dense(); //возвращает форму представления графа

bool Directed();//возвращает тип графа

double K(); //возвращает коэффициент насыщенности графа

void ToListGraph();//преобразует граф к L-графу

void ToMatrixGraph();//преобразует граф к M-графу

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

VertexT\* InsertV();//добавляет вершину к графу и возвращает адрес дескриптора вновь созданной вершины

bool DeleteV(VertexT \*v);//удаляет вершину из графа, заданную адресом дескриптора v

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

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

bool DeleteE(string v1, string v2);//удаляет ребро, соединяющее вершины, заданные адресами дескрипторов v1 и v2

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

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

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

bool hasEdge(int v1, int v2);

bool write\_name\_vertex(VertexT \*v1, string str);//Записать данные о вершине

bool write\_data\_vertex(VertexT \*v1, int d);//Записать данные о вершине

int read\_data\_vertex(VertexT \*v1);//Прочитать данные о вершине

void print\_graph();//Вывод на экран

int read\_weight\_edge(TVertex \*v1, TVertex \*v2);//Прочитать вес ребра

int read\_data\_edge(TVertex \*v1, TVertex \*v2);//Прочитать данные о ребре

bool write\_weight\_edge(TVertex \*v1, TVertex \*v2, int w);//Записать вес ребра

bool write\_data\_edge(TVertex \*v1, TVertex \*v2, int data);//Записать данные о ребре

VertexT\* getVertexFromName(string n);

VertexT\* getVertex(int index);

//Число вершин, рёбер, насыщенность

int getIndex(VertexT \*v);

//Итератор вершин

class VertexIterator

{

private:

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

bool end; //Состояние итератора

int current; //Текущее положение

public:

VertexIterator(Graph<VertexT, EdgeT> &g) { graph = &g; }; //Конструктор

//возвращает итератор, установленный на первую вершину графа

bool begin()

{

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

{

end = true;

return false;

}

current = 0;

end = false;

return true;

}

//Переход к следующей позиции

bool operator++()

{

try {

if (end) {

current = -1;

throw "Попытка выхода за пределы графа";

}

}

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()

{

end = true;

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

if (current < 0)

return false;

else

return true;

}

//Проверка на выход

bool onEnd()

{

return end;

}

//Получение вершины

VertexT\* operator\*()

{

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

throw "Нельзя обратиться к вершине";

return graph->vertexVector[current];

}

//Прочитать вершину по текущей позиции итератора

string read\_vertex()

{

VertexT \*v = NULL;

try

{

v = operator\*();

}

catch (...) { throw; }

if (v->isNameSet())

return v->getName();

else

throw "Имя вершины не установлено";

}

//Прочитать данные вершины по текущей позиции итератора

int read\_data\_vertex()

{

VertexT \*v = NULL;

try

{

v = operator\*();

}

catch (...) { throw; } {

if (v->isDataSet())

return v->getData();

else throw "Данные вершины не установлены";

}

}

//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 "Итератор за пределами графа";

if (useM)

return mIt->getEdge();

else

return lIt->getEdge();

}

//Прочитать вершины ребра и вес по текущей позиции итератора

string read\_edge()

{

EdgeT\* e = operator\*();

VertexT \*\_v1, \*\_v2;

\_v1 = e->getVertex1();

\_v2 = e->getVertex2();

if (!(\_v1->isNameSet()))

str1 = "Имя ребра не установлено";

else

str1 = \_v1->getName();

if (!(\_v2->isNameSet()))

str2 = "Данные ребра не установлены";

else

str2 = \_v2->getName();

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

str3 = "Вес ребра не установлен";

else

str3 = int\_to\_string(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 = operator\*();

if (f->isDataSet())

return f->getData();

else

throw "Данные ребра не установлены";

}

bool write\_data\_edge(int d)

{

EdgeT\* e;

try {

e = operator\*();

}

catch (char\* e) {

cerr << endl << e << endl;

return false;

}

e->setData(d);

return true;

}

bool write\_weight\_edge(int w)

{

EdgeT\* e;

try {

e = operator\*();

}

catch (char\* e) {

cerr << endl << e << endl;

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;

bool end;

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

graph = &g;

mIt = NULL;

lIt = NULL;

int curV = graph->getIndex(&v);

if (graph->Dense()) {

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

useM = true;

}

else {

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

useM = false;

}

}

~OutputEdgeIterator()

{

if (mIt != NULL)

delete mIt;

if (lIt != NULL)

delete lIt;

}

//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 "Итератор за пределами графа";

if (useM)

return mIt->getEdge();

else

return lIt->getEdge();

}

string read\_edge() {

EdgeT\* e = operator\*();

VertexT \*\_v1, \*\_v2;

\_v1 = e->getVertex1();

\_v2 = e->getVertex2();

if (!(\_v1->isNameSet()))

str1 = "Имя ребра не установлено";

else

str1 = \_v1->getName();

if (!(\_v2->isNameSet()))

str2 = "Данные ребра не установлены";

else

str2 = \_v2->getName();

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

str3 = "Вес ребра не установлен";

else

str3 = int\_to\_string(e->getWeight());

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

return str4;

}

int read\_data\_edge() {

EdgeT\* e;

e = operator\*();

if (e->isDataSet())

return e->getData();

else

throw "Данные ребра не установлены";

}

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\*e) {

cerr << endl << e << endl;

return false;

}

e->setWeight(w);

return true;

}

};

friend class VertexIterator; //дружественный класс

friend class EdgeIterator;

friend class OutputEdgeIterator;

};

//Получение индекса вершины по дескриптору

template <class VertexT, class EdgeT>

int Graph<VertexT, EdgeT>::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 "Вершина не найдена";

return index;

}

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

template <class VertexT, class EdgeT>

Graph<VertexT, EdgeT>::Graph()

{

sz = 0;

data = new GraphListForm<EdgeT>(false);

directed = false;

dense = false;

edgeCounter = 0;

}

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

template <class VertexT, class EdgeT>

Graph<VertexT, EdgeT>::Graph(int vertexCount, bool directed, bool dense)

{

sz = 0;

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(int\_to\_string(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

template <class VertexT, class EdgeT>

Graph<VertexT, EdgeT>::Graph(int vertexCount, int edgeCount, bool directed, bool dense)

{

sz = 0;

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(int\_to\_string(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((unsigned)time(NULL));

}

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

template <class VertexT, class EdgeT>

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

{

sz = 0;

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

template <class VertexT, class EdgeT>

Graph<VertexT, EdgeT>::~Graph()

{

while (vertexVector.size() != 0)

DeleteV(0);

delete data;

}

//возвращает число вершин в графе

template <class VertexT, class EdgeT>

int Graph<VertexT, EdgeT>::V()

{

return vertexVector.size();

}

//возвращает число ребер в графе

template <class VertexT, class EdgeT>

int Graph<VertexT, EdgeT>::E()

{

return edgeCounter;

}

//возвращает тип графа

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::Directed()

{

return directed;

}

//возвращает форму представления графа

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::Dense()

{

return dense;

}

//возвращает коэффициент насыщенности графа

template <class VertexT, class EdgeT>

double Graph<VertexT, EdgeT>::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

template <class VertexT, class EdgeT>

void Graph<VertexT, EdgeT>::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

template <class VertexT, class EdgeT>

void Graph<VertexT, EdgeT>::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

template <class VertexT, class EdgeT>

VertexT\* Graph<VertexT, EdgeT>::InsertV()

{

VertexT \*v = new VertexT;

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

throw 1;

v->setName(int\_to\_string(sz));

sz++;

vertexVector.push\_back(v);

return v;

}

//удаляет вершину из графа, заданную адресом дескриптора v

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::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;

}

template <class VertexT, class EdgeT>

VertexT\* Graph<VertexT, EdgeT>::getVertex(int index)

{

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

throw "Вершина не найдена";

return vertexVector[index];

}

template <class VertexT, class EdgeT>

int Graph<VertexT, EdgeT>::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 "Вершина не найдена";

return index;

}

template <class VertexT, class EdgeT>

EdgeT\* Graph<VertexT, 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;

}

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::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;

}

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::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));

}

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::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);

}

template <class VertexT, class EdgeT>

EdgeT\* Graph<VertexT, EdgeT>::getEdge(VertexT \*v1, VertexT \*v2)

{

EdgeT\* e;

try {

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

}

catch (int) {

throw "Ребро не найдено";

}

return e;

}

template <class VertexT, class EdgeT> //Записать данные о ребре

bool Graph<VertexT, EdgeT>::write\_data\_edge(TVertex\* v1, TVertex\* v2, int data)

{

EdgeT\* e;

try {

e = getEdge(v1, v2);

e->setData(data);

}

catch (char\*e) {

cerr << endl << e << endl;

return false;

}

return true;

}

//Записать вес ребра

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::write\_weight\_edge(TVertex\* v1, TVertex\* v2, int w)

{

EdgeT\* e;

try {

e = getEdge(v1, v2);

e->setWeight(w);

}

catch (char\*e) {

cerr << endl << e << endl;

return false;

}

return true;

}

//Прочитать данные о ребре

template <class VertexT, class EdgeT>

int Graph<VertexT, EdgeT>::read\_data\_edge(TVertex\* v1, TVertex\* v2)

{

EdgeT\* e;

try {

e = getEdge(v1, v2);

}

catch (char\*e) {

cerr << endl << e << endl;

return -1;

}

try {

if (e->isDataSet())

return e->getData();

else

throw "Данные ребра не установлены";

}

catch (char\*e) {

cerr << endl << e << endl;

return -1;

}

}

//Прочитать вес ребра

template <class VertexT, class EdgeT>

int Graph<VertexT, EdgeT>::read\_weight\_edge(TVertex\* v1, TVertex\* v2)

{

EdgeT\* e;

try {

e = getEdge(v1, v2);

}

catch (char\*e) {

cerr << endl << e << endl;

return -1;

}

try {

if (e->isWeightSet())

return e->getWeight();

else

throw "Вес ребра не установлен";

}

catch (char\*e) {

cerr << endl << e << endl;

return -1;

}

}

//Вывод на экран

template <class VertexT, class EdgeT>

void Graph<VertexT, EdgeT>::print\_graph()

{

data->print(vertexVector);

};

//Прочитать данные о вершине

template <class VertexT, class EdgeT>

int Graph<VertexT, EdgeT>::read\_data\_vertex(VertexT \*v)

{

int d = v->getData();

try {

if (v->isDataSet())

return d;

else

throw invalid\_argument("Нет данных для данной вершины");

}

catch (invalid\_argument &e) {

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

return -1;

}

}

//Записать данные о вершине

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::write\_data\_vertex(VertexT \*v1, int d)

{

v1->setData(d);

return true;

}

//Переименовать вершину

template <class VertexT, class EdgeT>

bool Graph<VertexT, EdgeT>::write\_name\_vertex(VertexT \*v1, string str)

{

v1->setName(str);

return true;

}

Task1.h

#include "Graph.h"

#include "Vertex.h"

#include "Edge.h"

#include <stack>

template <class VertexT, class EdgeT>

class Task1

{

private:

stack<string> st;

string resetVer;

bool DFS\_Visit\_Gamilton(string beg, string cur, int d)

{

TGraph::VertexIterator \*iter = new TGraph::VertexIterator(\*graph);

//cout << "cur = " << cur << " " << "d = " << d << "\n";

if (cur == beg)

{

if (d == 1)

{

return true;

}

}

TVertex \*ve = graph->getVertexFromName(cur);

ve->setData(1);

st.push(cur);

TGraph::OutputEdgeIterator \*oIt = new TGraph::OutputEdgeIterator(\*graph, \*(graph->getVertexFromName(cur)));

oIt->begin();

do

{

oIt->read\_edge();

ve = graph->getVertexFromName(oIt->str2);

if (ve->getData() == 0)

{

if (DFS\_Visit\_Gamilton(beg, oIt->str2, d - 1) == true)

return true;

}

} while (++\*oIt);

ve = graph->getVertexFromName(cur);

ve->setData(0);

st.pop();

return false;

}

bool DFS\_Gamilton(string v) // 0 - БЕЛЫЙ 1 - СЕРЫЙ

{

int Vnum = graph->V();

TGraph::VertexIterator \*iter = new TGraph::VertexIterator(\*graph);

while (!st.empty())

st.pop();

iter->begin();

do

{

iter->write\_data\_vertex(0);

} while (++\*iter);

Vnum = graph->V();

iter->begin();

TGraph::OutputEdgeIterator \*oIt = new TGraph::OutputEdgeIterator(\*graph, \*(graph->getVertexFromName(v)));

oIt->begin();

oIt->read\_edge();

st.push(v);

string str = oIt->str1;

do

{

oIt->read\_edge();

if (DFS\_Visit\_Gamilton(str,oIt->str2,Vnum) == true)

return true;

} while (++\*oIt);

return false;

}

public:

Graph<VertexT, EdgeT>\* graph;

Task1(Graph< VertexT, EdgeT> \*graph)

{

this->graph = graph;

}

Task1(Task1 \*obj)

{

this->st = obj->st;

this->graph = obj->graph;

this->resetVer = obj->resetVer;

}

stack<string> set(Graph<VertexT, EdgeT> \*g, string v)

{

resetVer = v;

graph = g;

DFS\_Gamilton(v);

return st;

}

stack<string> restart()

{

if (graph != NULL)

return set(graph, resetVer);

}

void result()

{

if (st.empty())

{

cout << "Задача не решена" << endl;

return;

}

stack<string> \*tmp = new stack<string>(st);

while (!tmp->empty())

{

cout << " <- " << tmp->top();

tmp->pop();

}

cout << endl;

delete tmp;

}

~Task1(){}

};

Task2.h

#pragma once

#include "Graph.h"

#include "Vertex.h"

#include "Edge.h"

#include <stack>

template <class VertexT, class EdgeT>

class Task2

{

private:

int \*\*matrix;

int res[2];

int \*d; // минимальное расстояние

int \*v; // посещенные вершины

int resetVer1;

int resetVer2;

void getMatrix() {

int sz = graph->V();

int vv1, vv2, vv3;

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

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

try {

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

}

catch (char \*) {

continue;

}

out->begin();

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 >> vv3;

if (u == vv2) {

vv2 = vv1;

}

matrix[u][vv2] = vv3;

try {

++\*out;

}

catch (char \*) {

break;

}

}

}

}

int deikstra(int v1, int v2) // возвращает кратчайшее расстояние из v1 в v2

{

int size = graph->V();

d = new int[size];

v = new int[size];

int temp;

int minindex, min;

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

{

d[i] = 10000;

v[i] = 1;

}

d[v1] = 0;

do {

minindex = 10000;

min = 10000;

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

{

if ((v[i] == 1) && (d[i]<min))

{

min = d[i];

minindex = i;

}

}

if (minindex != 10000)

{

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

{

if (matrix[minindex][i] > 0)

{

temp = min + matrix[minindex][i];

if (temp < d[i])

{

d[i] = temp;

}

}

}

v[minindex] = 0;

}

} while (minindex < 10000);

return d[v2];

}

public:

Graph<VertexT, EdgeT>\* graph;

Task2(Task2 \*obj)

{

this->matrix = obj->matrix;

this->graph = obj->graph;

this->resetVer1 = obj->resetVer1;

this->resetVer2 = obj->resetVer2;

this->res = obj->res;

}

Task2(Graph< VertexT, EdgeT> \*graph)

{

this->graph = graph;

}

int\* set(int v1, int v2)

{

if (graph == NULL)

return NULL;

resetVer1 = v1;

resetVer2 = v2;

int size = graph->V();

matrix = new int \*[size];

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

{

matrix[i] = new int[size];

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

matrix[i][j] = 0;

}

getMatrix();

cout << "matrix: \n";

for (int i = 0; i < graph->V(); i++)

{

for (int j = 0; j < graph->V(); j++)

cout << matrix[i][j] << ' ';

cout << '\n';

}

cout << '\n';

int startWay = deikstra(v1, v2);

int incWay = 0;

res[0] = -1;

res[1] = -1;

int tmpW = 0;

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

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

{

if (matrix[i][j] == 0)

continue;

tmpW = matrix[i][j];

matrix[i][j] = 0;

incWay = deikstra(v1, v2);

if (incay > startWay)

{

startWay = incWay;

res[0] = i;

res[1] = j;

}

matrix[i][j] = tmpW;

}

return res;

}

int\* restart()

{

return set(resetVer1, resetVer2);

}

void result()

{

cout << "Ребро между вершинами " << res[0] << " и " << res[1] << endl;

}

};

Main.cpp

#pragma once

#include <iostream>

#include <string>

#include <cstdlib>

#include <ctime>

#include <list>

#include <map>

#include "Vertex.h"

#include "Edge.h"

#include "Form.h"

#include "Graph.h"

#include "Task1.h"

#include "Task2.h"

using namespace std;

typedef Vertex <string, int> TVertex;

typedef Edge<TVertex, int, int> TEdge;

typedef Graph <TVertex, TEdge> TGraph;

void getmenu();

void CreateCustomGraph(TGraph\* &graph, TGraph::VertexIterator\* &vIt, TGraph::EdgeIterator\* &eIt, TGraph::OutputEdgeIterator\* &oIt)

{

graph = new TGraph(NULL, true, false); //List

vIt = new TGraph::VertexIterator(\*graph);

eIt = new TGraph::EdgeIterator(\*graph);

}

void ClearMemory(TGraph\* &graph, TGraph::VertexIterator\* &vIt, TGraph::EdgeIterator\* &eIt, TGraph::OutputEdgeIterator\* &oIt)

{

if (graph != NULL)

delete graph;

if (vIt != NULL)

delete vIt;

if (eIt != NULL)

delete eIt;

if (oIt != NULL)

delete oIt;

graph = NULL;

vIt = NULL;

eIt = NULL;

oIt = NULL;

}

void getmenu() {

cout << "Меню:" << endl;

cout << "Создание графов:" << endl;

cout << "\t0. Создать пустой L-граф с нулевым числом вершин и ребер" << endl;

cout << "\t1. Создать граф с V вершинами, без ребер, типа D, формы представления F" << endl;

cout << "\t2. Создать граф с V вершинами, с E случайными ребрами, типа D, формы F" << endl;

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

cout << "\t3. Распечатать граф" << endl;

cout << "\t4. Опрос числа вершин в графе" << endl;

cout << "\t5. Опрос числа ребер в графе" << endl;

cout << "\t6. Опрос типа графа" << endl;

cout << "\t7. Опрос формы представления графа" << endl;

cout << "\t8. Опрос коэффициента насыщенности графа" << endl;

cout << "\t9. Преобразовать граф к L-форме" << endl;

cout << "\t10. Преобразовать граф к M-форме" << endl;

cout << "\t11. Добавить вершину к графу" << endl;

cout << "\t12. Удалить вершину из графа" << endl;

cout << "\t13. Вставить ребро в граф" << endl;

cout << "\t14. Удалить ребро из графа" << endl;

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

cout << "\t15. Прочитать данные о вершине" << endl;

cout << "\t16. Записать данные о вершине" << endl;

cout << "\t17. Перезаписать имя вершины" << endl;

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

cout << "\t18. Прочитать вес ребра" << endl;

cout << "\t19. Прочитать данные о ребре" << endl;

cout << "\t20. Записать данные о ребре" << endl;

cout << "\t21. Записать вес ребра" << endl;

cout << "Итератор вершин:" << endl;

cout << "\t22. Установить итератор вершин на начальную вершину" << endl;

cout << "\t23. Установить итератор вершин на конечную вершину" << endl;

cout << "\t24. Переход к следующей позиции" << endl;

cout << "\t25. Прочитать имя вершины по текущей позиции итератора" << endl;

cout << "\t26. Прочитать данные вершины по текущей позиции итератора" << endl;

cout << "\t27. Записать данные вершины по текущей позиции итератора" << endl;

cout << "\t28. Записать имя вершины по текущей позиции итератора" << endl;

cout << "Итератор ребер:" << endl;

cout << "\t29. Установить итератор ребер на начальное ребро" << endl;

cout << "\t30. Установить итератор ребер на конечное ребро" << endl;

cout << "\t31. Переход к следующей позиции" << endl;

cout << "\t32. Прочитать ребро (вх., исх вершины, вес) по текущей позиции итератора" << endl;

cout << "\t33. Прочитать данные ребра по текущей позиции итератора" << endl;

cout << "\t34. Записать данные о ребре по текущей позиции итератора" << endl;

cout << "\t35. Записать вес ребра по текущей позиции итератора" << endl;

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

cout << "\t36. Создать итератор исходящих ребер" << endl;

cout << "\t37. Установить итератор исходящих ребер на начальное ребро" << endl;

cout << "\t38. Установить итератор исходящих ребер на конечное ребро" << endl;

cout << "\t39. Переход к следующей позиции" << endl;

cout << "\t40. Прочитать вершины ребра и вес по текущей позиции итератора" << endl;

cout << "\t41. Прочитать данные ребра по текущей позиции итератора" << endl;

cout << "\t42. Записать данные о ребре по текущей позиции итератора" << endl;

cout << "\t43. Записать вес ребра по текущей позиции итератора" << endl;

cout << "Задача 1 (Гамильтонов цикл)" << endl;

cout << "\t50) Выполнить" << endl;

cout << "\t51) Выполнить повторно" << endl;

cout << "\t52) Получить результат выполнения" << endl;

cout << "Задача 2 Нахождения ребра , устранение которого вызывает максимальное возрастание кратчайшего пути из вершины u\nв вершину v во взвешенном орграфе на основе алгоритма Дейкстры)" << endl;

cout << "\t55) Выполнить" << endl;

cout << "\t56) Выполнить повторно" << endl;

cout << "\t57) Получить результат выполнения" << endl;

cout << "99) Вывести список команд" << endl;

cout << "100) Выход" << endl;

cout << "99. Вывести список команд" << endl;

cout << "100. Выход" << endl;

}

int main(int argc, char\* argv[])

{

bool on = false; //Итератор исход реб

//Граф

TGraph\* graph = NULL;

//Итераторы

TGraph::VertexIterator \*vIt = NULL;

TGraph::EdgeIterator \*eIt = NULL;

TGraph::OutputEdgeIterator \*oIt = NULL;

map<string, TVertex\*> MAP;

CreateCustomGraph(graph, vIt, eIt, oIt);

Task1<TVertex, TEdge> \*task = new Task1<TVertex, TEdge>(graph);

Task2<TVertex, TEdge> \*task2 = new Task2<TVertex, TEdge>(graph);

bool isExit = false; //Флаг выхода

setlocale(LC\_ALL, "Russian");

getmenu();

while (!isExit)

{

int ans; //Код нажатой клавиши

cin >> ans;

if (graph == NULL && ans > 2 && ans < 100)

{

cout << "Граф не создан" << endl;

continue;

}

if (oIt == NULL && ans > 36 && ans < 44)

{

cout << "Итератор не создан" << endl;

continue;

}

try

{

switch (ans)

{

case 0://Создать пустой L-граф с нулевым числом вершин и ребер

{

MAP.clear();

ClearMemory(graph, vIt, eIt, oIt);

graph = new TGraph();

vIt = new TGraph::VertexIterator(\*graph);

eIt = new TGraph::EdgeIterator(\*graph);

cout << endl << "Граф создан" << endl;

break;

}

case 1: //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;

ClearMemory(graph, vIt, eIt, oIt);

graph = new TGraph(0, tmp2, tmp3);

TVertex \*v;

MAP.clear();

for (int i = 0; i < tmp1; i++)

{

v = graph->InsertV();

MAP.insert(pair<string, TVertex\*>(v->getName(), v));

}

cout << MAP.size() << endl;

for (int i = 0; i < tmp1; i++)

cout << MAP.at(to\_string(i))->getName() << endl;

vIt = new TGraph::VertexIterator(\*graph);

eIt = new TGraph::EdgeIterator(\*graph);

cout << endl << "Граф создан" << endl;

break;

}

case 2://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;

ClearMemory(graph, vIt, eIt, oIt);

graph = new TGraph(tmp1, tmp2, tmp3, tmp4);

vIt = new TGraph::VertexIterator(\*graph);

eIt = new TGraph::EdgeIterator(\*graph);

cout << endl << "Граф создан" << endl;

break;

}

case 3:

graph->print\_graph();

break;

case 4:

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

break;

case 5:

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

break;

case 6:

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

break;

case 7:

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

break;

case 8:

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

break;

case 9:

{

if (vIt != NULL)

delete vIt;

if (eIt != NULL)

delete eIt;

graph->ToListGraph();

vIt = new TGraph::VertexIterator(\*graph);

eIt = new TGraph::EdgeIterator(\*graph);

cout << "Граф преобразован" << endl;

break;

}

case 10:

{

if (vIt != NULL)

delete vIt;

if (eIt != NULL)

delete eIt;

graph->ToMatrixGraph();

vIt = new TGraph::VertexIterator(\*graph);

eIt = new TGraph::EdgeIterator(\*graph);

cout << "Граф преобразован" << endl;

break;

}

case 11:

{

TVertex \*v = graph->InsertV();

MAP.insert(pair<string, TVertex\*>(v->getName(), v));

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

break;

}

case 12:

{

string v1;

cout << "Введите удаляемую вершину: ";

cin >> v1;

TVertex \*v = MAP.at(v1);

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

MAP.erase(v1);

break;

}

case 13:

{

string v1, v2;

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

cin >> v1;

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

cin >> v2;

cout << "Ребро добавлено в граф (1 - да, 0 - нет): ";

try {

graph->InsertE(MAP.at(v1), MAP.at(v2));

}

catch (char\*) {

cout << "0" << endl;

break;

}

cout << "1" << endl;

break;

}

case 14:

{

string v1, v2;

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

cin >> v1;

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

cin >> v2;

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

break;

}

//Операции над вершинами

case 15://Прочитать данные о вершине

{

int tmp;

string v1;

cout << "Введите имя вершины: ";

cin >> v1;

TVertex \*v = MAP.at(v1);

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

if (tmp != -1)

cout << endl << "Данные вершины: " << tmp << endl;

break;

}

case 16://Записать данные о вершине

{

int tmp;

string v1;

cout << "Введите имя вершины: ";

cin >> v1;

cout << "Введите данные: ";

cin >> tmp;

TVertex \*v = MAP.at(v1);

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

break;

}

case 17://Записать новое имя вершины

{

string v1, v2;

cout << "Введите старое имя вершины: ";

cin >> v1;

cout << "Введите новое имя: ";

cin >> v2;

TVertex \*v = MAP.at(v1);

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

MAP.insert(pair<string, TVertex\*>(v2, v));

MAP.erase(v1);

break;

}

case 18://Прочитать вес ребра

{

int tmp;

string v1, v2;

cout << "Введите имя исходящей вершины: ";

cin >> v1;

cout << "Введите имя входящей вершины: ";

cin >> v2;

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

if (tmp != -1)

cout << "Вес ребра: " << tmp << endl;

break;

}

case 19://Прочитать данные о ребре

{

int tmp;

string v1, v2;

cout << "Введите имя исходящей вершины: ";

cin >> v1;

cout << "Введите имя входящей вершины: ";

cin >> v2;

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

if (tmp != -1)

cout << "Данные ребра: " << tmp << endl;

break;

}

case 20://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;

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

if (tmp1 == true)

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

break;

}

case 21://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;

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

if (tmp1 == true)

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

break;

}

//Итератор вершин

case 22://Установить итератор вершин на начальную вершину

{

if (!(vIt->begin()))

cout << "Итератор за пределами графа" << endl;

break;

}

case 23://Установить итератор вершин на конечную вершину

{

if (!(vIt->toend()))

cout << "Итератор за пределами графа" << endl;

break;

}

case 24://Переход к следующей позиции

{

bool tmp;

tmp = ++\*vIt;

break;

}

case 25://Прочитать имя вершины по текущей позиции итератора

{

cout << vIt->read\_vertex() << endl;

break;

}

case 26://Прочитать данные вершины по текущей позиции итератора

{

cout << vIt->read\_data\_vertex() << endl;

break;

}

case 27://Записать данные о вершины по текущей позиции итератора

{

int tmp;

bool tmp2;

cout << "Введите данные: ";

cin >> tmp;

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

break;

}

case 28://Записать имя вершины по текущей позиции итератора

{

string cur;

bool tmp;

cout << "Введите имя: ";

cin >> cur;

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

break;

}

//Итератор ребер

case 29://Установить итератор ребер на начальное ребро

{

if (eIt->begin())

cout << "Итератор установлен на начальное ребро" << endl;

break;

}

case 30://Установить итератор ребер на конечное ребро

{

if (eIt->toend())

cout << "Итератор установлен на конечное ребро" << endl;

break;

}

case 31://Переход к следующей позиции

{

if (!(++(\*eIt)))

cout << "Итератор за пределами графа" << endl;

break;

}

case 32://Прочитать вершины ребра и вес по текущей позиции итератора

{

cout << eIt->read\_edge() << endl;

break;

}

case 33://Прочитать данные ребра по текущей позиции итератора

{

cout << eIt->read\_data\_edge() << endl;

break;

}

case 34://Записать данные о ребре по текущей позиции итератора

{

int tmp;

cout << "Введите данные: ";

cin >> tmp;

eIt->write\_data\_edge(tmp);

break;

}

case 35://Записать вес ребра по текущей позиции итератора

{

int tmp;

cout << "Введите вес: ";

cin >> tmp;

eIt->write\_weight\_edge(tmp);

break;

}

case 36:

{

string v;

cout << "Введите имя вершины: ";

cin >> v;

oIt = new TGraph::OutputEdgeIterator(\*graph, \*(graph->getVertexFromName(v)));

cout << "Итератор создан" << endl;

break;

}

//Итератор исходящих ребер

case 37://Установить итератор исходящих ребер на начальное ребро

{

if (oIt->begin())

cout << "Итератор установлен на начальное ребро" << endl;

break;

}

case 38://Установить итератор исходящих ребер на конечное ребро

{

if (!(oIt->toend()))

cout << "Итератор установлен на конечное ребро" << endl;

break;

}

case 39://Переход к следующей позиции

{

if (!(++(\*oIt)))

cout << "Итератор за пределами графа" << endl;

break;

}

case 40://Прочитать вершины ребра и вес по текущей позиции итератора

{

cout << oIt->read\_edge() << endl;

break;

}

case 41://Прочитать данные ребра по текущей позиции итератора

{

int tmp;

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

cout << tmp << endl;

break;

}

case 42://Записать данные о ребре по текущей позиции итератора

{

int tmp;

cout << "Введите данные: ";

cin >> tmp;

oIt->write\_data\_edge(tmp);

break;

}

case 43://Записать вес ребра по текущей позиции итератора

{

int tmp;

cout << "Введите вес: ";

cin >> tmp;

oIt->write\_weight\_edge(tmp);

break;

}

case 50: //Выполнение задачи 1

{

string v;

cout << "Введите номер стартовой вершины " << endl;

cin >> v;

try

{

graph->getVertexFromName(v);

}

catch (char\*)

{

cout << "Некорректная вершина" << endl;

break;

}

task = new Task1<TVertex, TEdge>(graph);

task->set(graph, v);

break;

}

case 51: // Повторное выполнение задачи 1

{

task->restart();

break;

}

case 52: // Получение результата задачи 1

{

deque<string> st = task->result();

deque<string> \*tmp = new deque<string>(st);

while (!tmp->empty())

{

cout << " -> " << tmp->front();

tmp->pop\_front();

}

break;

}

case 55: // Выполнение задачи 2

{

task2 = new Task2<TVertex, TEdge>(graph);

string v1, v2;

cout << "Введите номер вершины u " << endl;

cin >> v1;

cout << "Введите номер вершины v " << endl;

cin >> v2;

try

{

graph->getVertexFromName(v1);

graph->getVertexFromName(v2);

}

catch (char\*)

{

cout << "Некорректная вершина" << endl;

break;

}

task2->set(atoi(v1.c\_str()), atoi(v2.c\_str()));

break;

}

case 56: // Повторное выполнение задачи 2

{

task2->restart();

break;

}

case 57: // Получение результата задачи 2

{

int \*buf = task2->result();

cout << "Ребро между вершинами " << buf[0] << " и " << buf[1] << endl;

break;

}

case 99://Вывод меню

{

getmenu();

break;

}

case 100: //Выход

isExit = true;

break;

case 101: //Создать готовый граф:

//Очистка памяти:

ClearMemory(graph, vIt, eIt, oIt);

graph = new TGraph(6, true, true); //List

graph->InsertE("0", "1");

graph->InsertE("0", "3");

graph->InsertE("0", "4");

graph->InsertE("1", "2");

graph->InsertE("2", "3");

graph->InsertE("3", "1");

graph->InsertE("4", "2");

graph->InsertE("4", "5");

/\*graph->write\_weight\_edge("0", "1", 3);

graph->write\_weight\_edge("0", "3", 11);

graph->write\_weight\_edge("1", "2", 1);

graph->write\_weight\_edge("2", "3", 3);

graph->write\_weight\_edge("3", "1", 8);

graph->write\_weight\_edge("4", "2", 4);

graph->write\_weight\_edge("4", "5", 6);\*/

//Ориентированный ли граф (false - нет, true - да)

//Форму графа (false - список, true - матрица)

vIt = new TGraph::VertexIterator(\*graph);

eIt = new TGraph::EdgeIterator(\*graph);

break;

default: //Нажата клавиша, не соответствующая ни одной команде

cout << "Неизвестная команда" << endl;

break;

}

}

catch (char\* err)

{

cout << err << endl;

}

catch (exception& exc)

{

cout << exc.what() << endl;

}

}

//Очистка памяти:

ClearMemory(graph, vIt, eIt, oIt);

return 0;

}