Файл Iterator.h

#pragma once

template<class T>

class Iterator

{

protected:

//флаг, показывающий, достиг ли итератор концв списка

//должен поддерживаться производными классами

int iterationComplete;

public:

//конструктор

Iterator() : iterationComplete{ 0 }

{}

//обязательные методы итератора

virtual void Next() = 0;

virtual void Reset() = 0;

//методы выборки/модификации данных

virtual T& Data() = 0;

//проверка конца списка

virtual int EndOfList() const {

return iterationComplete;

}

};

Файл List.h

#pragma once

template<class T>

class List // абстрактный класс

{

//число элементов списка, обновляемое производным классом

int size;

public:

//конструктор устанавливает size в 0

List()

{

this->size = 0;

}

//ввернуть размер списка

int ListSize() {

return this->size;

}

//проверить, пуст ли список

int ListEmpty() {

return this->size == 0;

}

//найти

virtual int Find(T& item) = 0;

//вставить элемент

virtual void Insert(const T& item) = 0;

//удалить элемент

virtual void Delete(const T& item) = 0;

//очистить список

virtual void ClearList() = 0;

};

Файл Node.h

#pragma once

template <typename T>

class Node // шаблонный класс узла списка

{

private:

Node\* next; // указатель на следующий узел

public:

T data; // данные узла списка

// конструктор

Node(const T& item, Node<T>\* ptrnext = nullptr);

// методы модификации списка

void InsertAfter(Node<T>\* p);

Node<T>\* DeleteAfter();

// получение адреса следующего узла

Node<T>\* NextNode() const;

};

template <class T>

Node<T>::Node(const T& item, Node<T>\* ptrnext) : data{ item }, next{ ptrnext } {};

// возвратить заркытый член next

template <class T>

Node<T>\* Node<T>::NextNode() const {

return next;

}

// вставить узел p после текущего узла

template <class T>

void Node<T>::InsertAfter(Node<T>\* p) {

// p указывает на следующий за текущим узел

// а текущий узел - на p

p->next = next;

next = p;

}

// удалить узел, следующий за текущим, и возвратить его адрес

template <class T>

Node<T>\* Node<T>::DeleteAfter()

{

// сохранить адрес удаляемого узла

Node<T>\* tempPtr = next;

// если нет следующего узла, возвратить NULL

if (next == nullptr)

return nullptr;

// текущий узел указывает на узел, следующий за tempPtr

next = tempPtr->next;

// возвратить указатель на несвязанный узел

return tempPtr;

}

Файл LinkedList.h

#pragma once

#include <iostream>

#include <stdlib.h>

#include "Node.h"

using namespace std;

template <class T>

class LinkedList

{

private:

// указатели для доступа к началу и концу списка

Node<T>\* front, \* rear;

// используются для извлечения, вставки и удаления данных

Node<T>\* prevPtr, \* currPtr;

// число элементов в списке

int size;

// положение в списке. используется методом Reset

int position;

// создание узла

Node<T>\* GetNode(const T& item, Node<T>\* ptrNext = nullptr);

// очистка узла

void FreeNode(Node<T>\* pl);

// колирует список L в текущий список

void CopyList(const LinkedList<T>& L);

public:

// конструктор

LinkedList();

// конструктор

LinkedList(const LinkedList<T>& L);

// деструктор

~LinkedList();

// проверка состояния списка

int ListSize() const;

// проверка на пустоту

int ListEmpty() const;

// установление position в положение pos

void Reset(int pos = 0);

// переход к следующему узлу

void Next();

// проверка, дошли ли до конца списка

bool EndOfList() const;

// возврат позиции

int CurrentPosition() const;

// вставить вначале

void InsertFront(const T& item);

// вставить в конец

void InsertRear(const T& item);

// вставить на какую-то позицию

void InsertAt(const T& item);

//вставить после

void InsertAfter(const T& item);

// методы удаления

T DeleteFront();

// удаление элемента

void DeleteAt();

// возвратить/изменить данные

T& Data();

// очистка списка

void ClearList();

LinkedList<T>& operator = (const LinkedList<T>& L);

};

// создание узла

template <class T>

Node<T>\* LinkedList<T>::GetNode(const T& item, Node<T>\* ptrNext) {

Node<T>\* temp = new Node<T>(item, ptrNext);

return temp;

};

// очистка узла

template <class T>

void LinkedList<T>::FreeNode(Node<T>\* pl) {

delete pl;

};

// колирует список L в текущий список

template <class T>

void LinkedList<T>::CopyList(const LinkedList<T>& L) {

// р - указатель на L

Node<T>\* p = L.front;

int pos;

// вставлять каждый элемент из L в конец текущего объекта

while (p != nullptr) {

InsertRear(p->data);

p = p->NextNode();

}

// выход, если список пустой

if (position == -1)

return;

// nереустановить prevPtr и currPtr в новом списке

prevPtr = nullptr;

currPtr = front;

for (pos = 0; pos != position; pos++) {

prevPtr = currPtr;

currPtr = currPtr->NextNode();

}

};

// конструктор

template <class T>

LinkedList<T>::LinkedList() {

front = nullptr;

rear = nullptr;

prevPtr = nullptr;

currPtr = nullptr;

size = 0;

position = 0;

};

// конструктор

template <class T>

LinkedList<T>::LinkedList(const LinkedList<T>& L) {

CopyList(L);

};

// деструктор

template <class T>

LinkedList<T>::~LinkedList(void) {

delete front;

delete rear;

};

// оператор присваивания

template <class T>

LinkedList<T>& LinkedList<T>::operator = (const LinkedList<T>& L) {

CopyList(L); //копируем список из входного

return \*this; //возвращаем этот список

};

// проверка состояния списка

template <class T>

int LinkedList<T>::ListSize(void) const {

return size;

};

// проверка на пустоту

template <class T>

int LinkedList<T>::ListEmpty(void) const {

return size == 0;

};

// установление position в положение pos

template <class T>

void LinkedList<T>::Reset(int pos) {

int startPos;

// если список пустой, выход

if (front == nullptr)

return;

// если положение задано не верно, закончить программу

if (pos < 0 || pos > size - 1)

{

cerr << "Reset: Неверно задано положение: " << pos << std::endl;

return;

}

// установить механизм nрохоЖдения в pos

if (pos == 0) {

// перейти в начало списка

prevPtr = nullptr;

currPtr = front;

position = 0;

}

else

// nереустановить currPtr, prevPtr, и startPos

{

currPtr = front->NextNode();

prevPtr = front;

startPos = 1;

// передвигаться вправо до pos

for (position = startPos; position != pos; position++) {

// передвинуть оба указателя прохождения вперед

prevPtr = currPtr;

currPtr = currPtr->NextNode();

}

}

};

// переход к следующему узлу

template <class T>

void LinkedList<T>::Next(void) {

// выйти, если конец списка или // список пустой

if (currPtr != nullptr) {

// nереустановить два указателя на один узел вперед

prevPtr = currPtr;

currPtr = currPtr->NextNode();

position++;

}

};

// проверка, дошли ли до конца списка

template <class T>

bool LinkedList<T>::EndOfList() const {

return position == size;

};

// возврат позиции

template <class T>

int LinkedList<T>::CurrentPosition(void) const {

return position;

};

// вставить вначале

template<typename T>

void LinkedList<T>::InsertFront(const T& item) {

Node<T>\* newNode; //новый узел

newNode = GetNode(item, front); //передаём в него значение и адрес начала

front = newNode; //указываем на новый узел

currPtr = front; //обновляем текущий указатель

position = 0; //обновляем позицию

size++; //увеличиваем размер

}

// вставить в конец

template<typename T>

void LinkedList<T>::InsertRear(const T& item) {

Node<T>\* newNode; //новый узел

this->currPtr = this->front;

this->position = 0;

//если список пуст, вставить item в начало

if (this->currPtr == nullptr)

this->InsertFront(item);

else {

//найти узел с нулевым указателем

while (this->currPtr->NextNode() != nullptr)

this->currPtr = this->currPtr->NextNode();

// создать узел и вставить в конец списка (после currPtr)

newNode = this->GetNode(item, this->rear);

this->currPtr->InsertAfter(newNode);

this->size++;

}

}

// вставить на какую-то позицию

template <class T>

void LinkedList<T>::InsertAt(const T& item) {

Node<T>\* newNode;

// два случая: вставка в начало или внутрь списка

if (prevPtr == nullptr) {

// вставка в начало списка. nоыещает также узел в пустой список

newNode = GetNode(item, front);

front = newNode;

}

else {

// вставка внутрь списка. помещает узел после prevPtr

newNode = GetNode(item);

prevPtr->InsertAfter(newNode);

}

// при prevPtr == rear, имеем вставку в пустой список

// или в хвост нелустого списка; обновляет rear и positioп

if (prevPtr == rear) {

rear = newNode;

position = size;

}

// обновить currPtr и увеличить size

currPtr = newNode;

size++;

};

//вставить после

template<typename T>

void LinkedList<T>::InsertAfter(const T& item) {

Node<T>\* newNode; //новый узел

if (this->front == nullptr) { //если список пуст

//вставляем его в начало

newNode = this->getNode(item, this->front);

this->front = newNode;

}

else {

if (this->currPtr == this->rear) { //если это хвостовой узел

//вставляем в конец списка

newNode = this->getNode(item, this->rear);

this->prev\_ptr->insAfter(newNode);

}

else {

//вставляем после текущего

newNode = this->getNode(item, currPtr->nextNode());

this->currPtr->insAfter(newNode);

}

}

this->next(); //смещаем позицию

this->size++;

}

//удаление

template <class T>

T LinkedList<T>::DeleteFront() {

Node<T>\* p;

T tmp = nullptr;

// ошибка, если список пустой или конец списка

if (currPtr == nullptr) {

cerr << "Ошибка удаления!" << std::endl;

exit(1);

}

if (front != nullptr)

{

p = front;

front = front->NextNode();

tmp = p->data;

}

FreeNode(p);

return tmp;

};

// удаление элемента

template <class T>

void LinkedList<T>::DeleteAt(void) {

Node<T>\* p;

// ошибка, если список пустой или конец списка

if (currPtr == nullptr) {

cerr << "Ошибка удаления!" << std::endl;

exit(1);

}

// удалять можно только в начале и внутри списка

if (prevPtr == nullptr) {

// сохранить адрес начала, но не связывать его. если это - последний узел, присвоить front значение NULL

p = front;

front = front->NextNode();

}

else {

// не связывать внутренний узел после prevPtr, запомнить адрес

p = prevPtr->DeleteAfter();

}

// если хвост удален, адрес нового хвоста в prevPtr, // а position уыеньwается на 1

if (p == rear) {

rear = prevPtr;

position--;

}

// установить currPtr на последний удаленный узел. Если р - последний узел в списке, currPtr становится равным NULL

currPtr = p->NextNode();

// освободить узел и уменьшить значение size

FreeNode(p);

size--;

};

// возвратить/изменить данные

template <class T>

T& LinkedList<T>::Data(void) {

// ошибка, если список пустой или прохождение закончено

if (size == 0 || currPtr == nullptr) {

cerr << "Data: Неверная ссылка!" << std::endl;

exit(1);

}

return currPtr->data;

};

// очистка списка template <class T>

void LinkedList<T>::ClearList() {

Node<T>\* currPosition, \* nextPosition;

currPosition = front;

while (currPosition != nullptr) {

// получить адрес следующего узла и удалить текущий

nextPosition = currPosition->NextNode();

FreeNode(currPosition);

currPosition = nextPosition; // перейти к следующему узлу

}

front = rear = nullptr;

prevPtr = currPtr = nullptr;

size = 0;

position = -1;

};

Файл SeqList.h

#pragma once

#include <iostream>

#include"LinkedList.h"

#include "List.h"

template<class T>

class SeqListIterator;

template<class T>

class SeqList:public List<T>

{

private:

LinkedList<T> llist;

public:

//конструктор

SeqList();

// методы доступа

int ListSize() const;

int ListEmpty() const;

int Find(const T& item);

T GetData(int pos);

// методы модификации

void Insert(const T& item);

void Delete(const T& item);

T DeleteFront();

void ClearList();

friend class SeqListIterator<T>;

};

//конструктор

template<class T>

SeqList<T>::SeqList() : List<T>(), llist() {

}

//поиск

template<class T>

int SeqList<T>::Find(const T& item) {

for(llist.Reset();!llist.EndOfList(); llist.Next())

if (item == llist.Data()) {

return 1;

}

return 0;

}

//получение данных

template<class T>

T SeqList<T>::GetData(int pos) {

if (pos < 0 || pos >= llist.ListSize()) {

cerr << "pos вне диапозона!" << endl;

exit(1);

}

llist.Reset(pos);

return llist.Data();

}

template<class T>

int SeqList<T>::ListSize() const {

return llist.ListSize();

}

template<class T>

int SeqList<T>::ListEmpty() const {

return llist.ListEmpty();

}

template<class T>

void SeqList<T>::Insert(const T& item) {

llist.InsertRear(item);

}

template<class T>

void SeqList<T>::Delete(const T& item) {

for (llist.Reset(); !llist.ListEmpty(); llist.Next())

if (item == llist.Data())

llist.DeleteAt();

}

template<class T>

T SeqList<T>::DeleteFront() {

return llist.DeleteFront();

}

template <class T>

void SeqList<T>::ClearList() {

llist.ClearList();

}

Файл SeqListIterator.h

#pragma once

#include <iostream>

#include "SeqList.h"

#include "Iterator.h"

template <class T>

class SeqListIterator : public Iterator<T> {

SeqList<T>\* listPtr; //локальный указатель на объект SeqList

Node<T>\* prevPtr, \* currPtr; //предыдущая и текущая позиция

public:

//конструктор

SeqListIterator(SeqList<T>& lst) : Iterator<T>(){

listPtr = &lst;

this->iterationComplete = listPtr->llist.ListEmpty();

Reset();

}

//продвинуться к следующему элементу списка

virtual void Next() {

if (currPtr == NULL) //currPtr == nullptr, мы в конце списка

return;

prevPtr = currPtr; //передвинуть указатель prevPtr на один узел вперед

currPtr = currPtr->NextNode(); //currPtr на один узел вперед

if (currPtr == NULL) //если обнаружен конец связанного списка

this->iterationComplete = 1; //установить флаг "итерация завершена"

}

//переход к началу списка

virtual void Reset() {

this->iterationComplete = listPtr->llist.ListEmpty(); //переприсвоить состояние итерации

if (listPtr->llist.front == nullptr) //вернуться если список пуст

return;

prevPtr = nullptr; //установить механизмы прохождения списка с первого узла

currPtr = listPtr->llist.front;

}

//возвратить данные, расположенные в текущем элементе списка

virtual T& Data() {

if (listPtr->llist.ListEmpty() || currPtr == nullptr) { //если список пуст или прохождение уже завершено

std::cerr << "Data: недопустимая ссылка!" << std::endl; //возвращаем ошибку

exit(1);

}

return currPtr->data;

}

//сейчас итератор должен проходить список lst

//переназначьте listPtr и вызовите reset

void SetList(SeqList<T>& lst) {

listPtr = &lst;

Reset(); //инициализировать механизм прохождения для списка lst

}

};

Файл PQueue.h

#pragma once

#include<iostream>

const int MAXSIZE = 50;

template <typename T>

class PQueue

{

T\* \_pqlist; // массив

int \_count; // кол-во элементов

public:

//конструктор

PQueue() {

this->\_pqlist = new T[MAXSIZE];

this->\_count = 0;

}

//кол-во элементов в списке

int lenght() const {

return this->\_count;

}

//проверка на пустоту

bool empty() const {

if (this->\_count == 0)

return true;

else

return false;

}

//удаляет элемент из очереди приоритетов

//и возвращает его значение

T del() {

if (empty() == false)

{

//найти минимальное значение и его индекс в массиве pqlist

T min = \_pqlist[0]; //предпологаем что pqlist[0] - это минимум

int tindex = 0;

//просмотреть остальные элементы

//измеряя минимум и его индекс

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

{

if (\_pqlist[i] < min) {

//новый минимум в элементе pqlist[i], новый индекс - i

min = \_pqlist[i];

tindex = i;

}

}

//переместить хвостовой элемент на место минимального

//и уменьшить на единицу count

this->\_pqlist[tindex] = this->\_pqlist[this->\_count-1];

\_count--;

return min;

}

else

std::cerr << "Queue is empty!" << std::endl;

}

//вставить элемент в очередь приоритетов

void insert(const T &item) {

//если уже все элементы массива pqlist использованы

//завершить программу

if (\_count == MAXSIZE) {

std::cerr << "Queue if full!" << std::endl;

exit(1);

}

//поместить элемент в конец списка

//увеличить count на единицу

this->\_pqlist[this->\_count] = item;

this->\_count++;

}

//удаление всех элементов

void clear() {

std::memset(&(this->\_pqlist[0]), 0, 50);

this->\_count = 0;

}

//деструктор

~PQueue() {

delete[]this->\_pqlist;

}

};

Файл Graph.h

#pragma once

#include "SeqList.h"

#include "SeqListIterator.h"

#include <stack>

#include "PQueue.h"

const int MAXGRAPHSIZE = 25;

template<class T>

class VertexIterator;

template<class T>

class Graph

{

private:

SeqList<T> vertexList;

int edge[MAXGRAPHSIZE][MAXGRAPHSIZE];

int graphsize;

int FindVertex(SeqList<T>& L, const T& vertex);

int GetVertexPos(const T& vertex);

public:

// конструктор

Graph();

// методы тестирования графа

int GraphEmpty() const;

int GraphFull() const;

// методы обработки данных

int NumberOfVertices() const;

int NumberOfEdges() const;

int GetWeight(const T& vertex1, const T& vertex2);

SeqList<T>& GetNeighbors(const T& vertex);

// методы модифицкации графа

void InsertVertex(const T& vertex);

void InsertEdge(const T& vertex1, const T& vertex2, int weight);

void DeleteVertex(const T& vertex);

void DeleteEdge(const T& vertex1, const T& vertex2);

// утилиты

void ReadGraph(const char\* filename);

int MinimumPath(const T& sVertex, const T& eVertex);

SeqList<T>& DepthFirstSerch(const T& beginVertex);

SeqList<T>& BreadthFirstSearch(const T& beginVertex);

// итератор для обхода вершин

friend class VertexIterator<T>;

};

//проверяет наличие вершины в списке L и исп в поисковых методах

template<class T>

int Graph<T>::FindVertex(SeqList<T>& L, const T& vertex) {

return L.Find(vertex);

}

// конструктор. обнуляет матрицу смежности и переменную graphsize

template<class T>

Graph<T>::Graph() : vertexList() {

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

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

edge[i][j] = 0;

graphsize = 0;

}

//сканирование списка вершин и возвращение позиции вершины в этом списке

template<class T>

int Graph<T>::GetVertexPos(const T& vertex) {

SeqListIterator<T> liter(vertexList);

int pos = 0;

while (!liter.EndOfList() && liter.Data() != vertex) {

pos++;

liter.Next();

}

return pos;

}

//возвращает вес ребра, которое соединяет vertex1 и vertex2

template<class T>

int Graph<T>::GetWeight(const T& vertex1, const T& vertex2) {

int i = GetVertexPos(vertex1);

int j = GetVertexPos(vertex2);

if (GetVertexPos(vertex1))

if (edge[i][j] > 0)

return edge[i][j];

else

return -1;

}

//возвращает список смежных вершин

template<class T>

SeqList<T>& Graph<T>::GetNeighbors(const T& vertex) {

SeqList<T>\* L;

SeqListIterator<T> viter(vertexList);

//создать пустой список

L = new SeqList<T>;

//позиция в списке, соответствующая номеру строки матрицы смежности

int pos = GetVertexPos(vertex);

//если такой вершины vertex нет в списке вершин, закончить работу программы

if (pos == -1) {

cerr << "GetNeighbors: такой вершины нет в графе." << endl;

return \*L;

}

//сканироватт строку матрицы смежности и включить в список

//все вершины, имеющие ребро ненулевого веса из vertex

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

if (edge[pos][i] > 0)

L->Insert(viter.Data());

viter.Next();

}

return \*L;

}

//удалить вершину из списка вершин и скорректировать матрицу смежности,

//удалив принадлежащие этой вершине ребра

template<class T>

void Graph<T>::DeleteVertex(const T& vertex) {

//получить позицию вершины в списке вершин

int pos = GetVertexPos(vertex);

int row, col;

//если такой вершины нет, сообщить об этом и вернуть управление

if (pos == -1) {

cerr << "DeleteVertex: у вершиные нет графы" << endl;

return;

}

//удалить вершину и уменьшить graphsize

vertexList.Delete(vertex);

graphsize--;

//матрица смежности делится на три области

for (row = 0; row < pos; row++) //область I

for (col = pos + 1; col < graphsize; col++)

edge[row][col - 1] = edge[row][col];

for (row = pos+1; row < graphsize; row++) //область II

for (col = pos + 1; col < graphsize; col++)

edge[row-1][col - 1] = edge[row][col];

for (row = pos+1; row < graphsize; row++) //область III

for (col = 0; col < pos; col++)

edge[row-1][col] = edge[row][col];

}

//начиная с начальной вершины, сформировать список вершин,

//обрабатываемых в порядке обхода "сначала в глубину"

template <class T>

SeqList<T>& Graph<T>::DepthFirstSerch(const T& beginVertex) {

//стек для временного хранения вершин, ожидающих обработки

stack<T> S;

//L - список пройденных вершин. adjL содержит вершины,

//смежные с текущей. L создается динамически, поэтому можно

//поэтому можно возвратить его адрес

SeqList<T>\* L, adjL;

SeqListIterator<T> iteradjL(adjL);

T vertex;

//инициализировать выходной список

//поместить начальную вершину в стек

L = new SeqList<T>;

S.push(beginVertex);

//продолжить сканирование, пока не опустеет стек

while (!S.empty()) {

//вытолкнуть очетедную вершину

vertex = S.top();

S.pop();

//проверить ее наличие в списке L

if (!FindVertex(\*L, vertex)) {

//если ее нет, включить вершину в L

//а также получить все смежные с ней вершины

(\*L).Insert(vertex);

adjL = GetNeighbors(vertex);

//установить итератор на текуший adjL

iteradjL.SetList(adjL);

//сканировать список смежных вершин

//помещать в стек те из них, которые отсутствуют в списке L

for (iteradjL.Reset(); !iteradjL.EndOfList(); iteradjL.Next())

if (!FindVertex(\*L, iteradjL.Data()))

S.push(iteradjL.Data());

}

}

//возвратить выходной список

return \*L;

}

//объекты типа PathInfo описывают путь, существующий между

//двумя вершинами и его стоимость

template<class T>

struct PathInfo

{

T startV, endV;

int cost;

};

//перегрузка оператора<=

template<class T>

int operator <= (const PathInfo<T>& a, const PathInfo<T> b)

{

return a.cost <= b.cost;

}

//минимальный путь

template <class T>

int Graph<T>::MinimumPath(const T& sVertex, const T& eVertex) {

//очедь приоритетов, в которую помещаются объекты,

//несущие информацию о стоимости путей из sVertex

PQueue<PathInfo<T>> PQ(MAXGRAPHSIZE);

//используется при вставке/удалении объектов PathInfo в очереди приоритетов

PathInfo<T> pathData;

//L - список всех вершин, достижимых из sVertex и стоимость которых уже учтена

//adjL - список вершин, смежных с посещаемой на данный момент

//для сканирование adjL использутся итератор adjLiter

SeqList<T>L, adjL;

SeqListIterator<T> adjLiter(adjL);

T sv, ev;

int mincost;

//сформировать начальный элемент очереди приоритетов

pathData.startV = sVertex;

pathData.endV = sVertex;

//стоимость пути из sVertex в eVertex равна 0

pathData.cost = 0;

PQ.insert(pathData);

//обрабатывать вершины, пока не будет найден минимальный путь к eVertex

//или пока не опустеет очередь приоритетов

while (!PQ.empty()) {

//удалить элемент приоритетной очереди и запомнить

//его конечную вершину и стоимость от sVertex

pathData = PQ.del();

ev = pathData.endV;

mincost = pathData.cost;

//если это eVertex

//то минимальный путь от sVertex к eVertex найден

if (ev == eVertex)

break;

//если конечная вершина уже имеется в L,

//не рассматривать ее снова

if (!FindVertex(L, ev)) {

//включить ev в список L

L.Insert(ev);

//найти все смежные с ev вершины. для тех из них,

//которых нет в L, сформировать объекты PathInfo с начальными

//вершинами, равнными ev, и включить их в очередь приоритетов

sv = ev;

adjL = GetNeighbors(sv);

//новый список adjL сканируется итератором adjLiter

adjLiter.SetList(adjL);

for (adjLiter.Reset(); !adjLiter.EndOfList(); adjLiter.Next()) {

ev = adjLiter.Data();

if (!FindVertex(L, ev)) {

//создать новый элемент приоритетной очереди

pathData.startV = sv;

pathData.endV = ev;

//стоимость равна текущей минимальной стоимости

//плюс стоимость от перехода от sv к ev

pathData.cost = mincost + GetWeight(sv, ev);

PQ.insert(pathData);

}

}

}

}

if (ev == eVertex)

return mincost;

else

return -1;

}

//граф пуст

template<class T>

int Graph<T>::GraphEmpty() const {

return graphsize == 0;

}

//граф забит

template<class T>

int Graph<T>::GraphFull() const {

return graphsize == MAXGRAPHSIZE;

}

//обращение к graphsize

template<class T>

int Graph<T>::NumberOfVertices() const {

return graphsize;

}

//кол-во ребер

template<class T>

int Graph<T>::NumberOfEdges() const {

int numOfEdges = 0;

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

for (int j = i; j < MAXGRAPHSIZE; j++)

if (edge[i][j] > 0)

numOfEdges++;

return numOfEdges;

}

//вставить новую вершину в множество вершин

template<class T>

void Graph<T>::InsertVertex(const T& vertex) {

vertexList.Insert(vertex);

graphsize++;

}

//вставить ребро (vertex1, vertex2) с весом weight в множество ребер

template<class T>

void Graph<T>::InsertEdge(const T& vertex1, const T& vertex2, int weight) {

if (vertex1 == nullptr && vertex2 == nullptr)

{

cout << "InsertEdge: ошибка вставки ребра!";

exit(1);

}

if(!vertexList.Find(vertex1))

{

vertexList.Insert(vertex1);

}

if (!vertexList.Find(vertex2))

{

vertexList.Insert(vertex2);

}

int i = GetVertexPos(vertex1);

int j = GetVertexPos(vertex2);

edge[i][j] = weight;

edge[j][i] = weight;

}

//если ребро (vertex1, vertex2) существует, удалить его из множества ребер

template <class T>

void Graph<T>::DeleteEdge(const T& vertex1, const T& vertex2) {

if (vertex1 != nullptr && vertex2 != nullptr && vertexList.Find(vertex1) && vertexList.Find(vertex2))

{

int i = GetVertexPos(vertex1);

int j = GetVertexPos(vertex2);

edge[i][j] = 0;

}

}

//параметр - имя файла с входным описанием вершин и ребер графа

template <class T>

void Graph<T>::ReadGraph(const char\* filename) {

}

template <class T>

SeqList<T>& Graph<T>::BreadthFirstSearch(const T& beginVertex) {

}

Файл VertexIterator.h

#pragma once

#include "SeqListIterator.h"

#include "Graph.h"

//доступ к компонентам графа

template<class T>

class VertexIterator : public SeqListIterator<T>

{

public:

VertexIterator(Graph<T>& G) : SeqListIterator<T>(G.vertexList) {};

};

Файл lab13.cpp

#include <iostream>

#include "Graph.h"

#include "VertexIterator.h"

//проверяет существование направленного пути от вершины v к вершине w и возвращает тру или фолс

template <class T>

int PathConnect(Graph<T>& G, T v, T w) {

SeqList<T> L;

//найти вершины, связанные с v

L = G.DepthFirstSerch(v);

//если w в их числе нет, вернуть тру

if (L.Find(w))

return 1;

else

return 0;

}

//печать сильной компоненты

template <class T>

void PrintList(SeqList<T> &list) {

SeqListIterator<T> liter(list);

for (liter.Reset(); !liter.EndOfList(); liter.Next()) {

cout << liter.Data() << " ";

}

}

template <class T>

void ConnectedComponent(Graph<T>& G) {

VertexIterator<T> viter(G);

SeqList<T> markedList, scList, L, R;

for (viter.Reset(); !viter.EndOfList(); viter.Next()) {

//проверять в цикле каждую вершину viter.Data()

//если не помечен, включить в сильную компоненту

if (!markedList.Find(viter.Data())) {

scList.ClearList();

//получить вершины, достижимые из viter.Data()

L = G.DepthFirstSerch(viter.Data());

//искать в списке вершины, из котрых достижима вершина viter.Data()

SeqListIterator<T> liter(L);

for(liter.Reset(); !liter.EndOfList(); liter.Next())

if (PathConnect(G, liter.Data(), viter.Data())) {

//вставить вершины в текущую сильную компоненту и в markedList

scList.Insert(liter.Data());

markedList.Insert(liter.Data());

}

PrintList(scList); //печать сильной компоненты

cout << endl;

}

}

}

int main()

{

setlocale(0, "ru");

Graph<const char\*> G;

G.InsertVertex("A");

G.InsertVertex("B");

G.InsertVertex("C");

G.InsertVertex("D");

G.InsertEdge("A", "B", 100);

G.InsertEdge("A", "C", 300);

G.InsertEdge("B", "C", 50);

G.InsertEdge("B", "D", 10);

G.InsertEdge("C", "D", 30);

cout << "Сильные компоненты:" << endl;

ConnectedComponent(G);

}

Результат

