Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №7

по теме «Обход графа в глубину»

Выполнили:

Студенты группы 21ВВ2

Назаров Е.А.

Макаров И.С.

Принял:

Митрохин М.А.

Юрова О.В.

Пенза, 2022

**Цель работы:**

Изучить основные алгоритмы обхода графа и научиться решать задачи обхода графа на основе поиска в глубину.

**Лабораторные работы:**

**Задание 1**

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран.
2. Для сгенерированного графа осуществите процедуру обхода в глубину, реализованную в соответствии с приведенным выше описанием.

**3.**\* Реализуйте процедуру обхода в глубину для графа, представленного списками смежности.

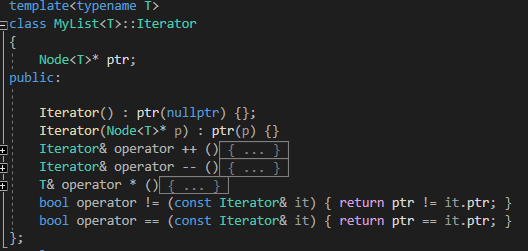
**Задание 2\***

1. Для матричной формы представления графов выполните преобразование рекурсивной реализации обхода графа к не рекурсивной.

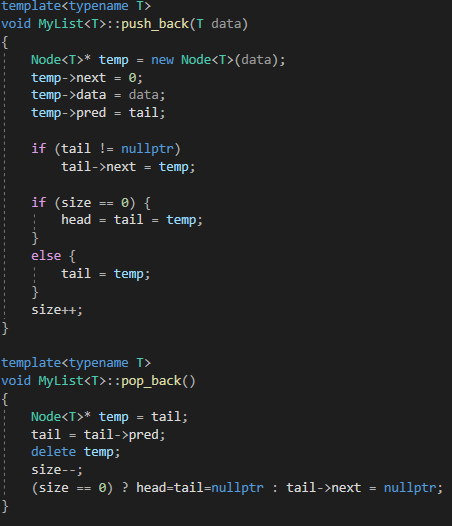
**Пояснительный текст к программе:**



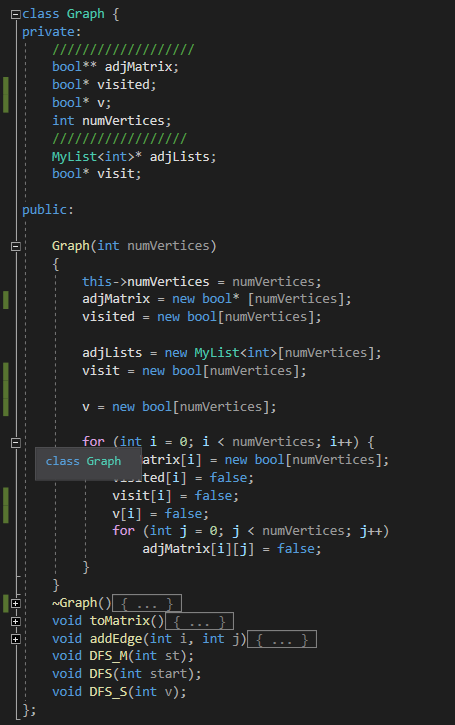
Был реализован динамический шаблонный двухсвязный список. Класс MyList представляет собой целостный класс с описанными функциями в нем. В классе был реализован класс Node элемента, содержащий в себе всю информацию и класс Iterator для перемещения по списку.



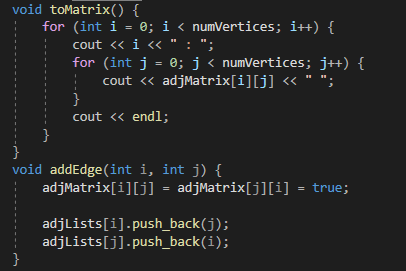
Содержит в себе в большей степени операторов перегрузки для хождения от одного элемента класса к другому. Функции, возвращающие начало и конец были вынесены в класс MyList.



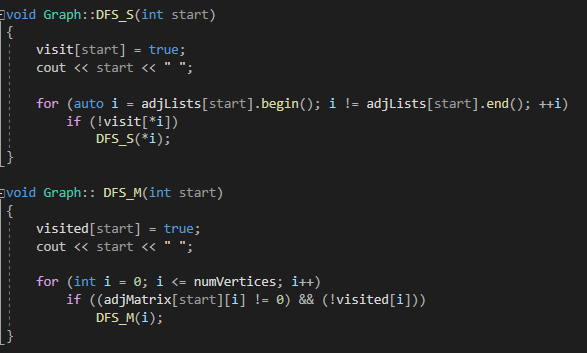
Функции добавления в конец списка(push\_back) и удаления последнего элемента списка(pop\_back).



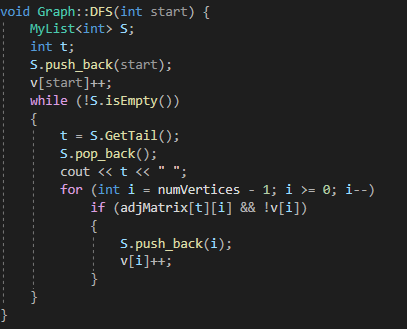
Класс содержащий в себе информацию о графе. В конструктор поступает количество вершин, после чего выделяется динамическая память для всех нужных элементов и массивов. Деструктор же освобождает эту память в конце программы.



Функции вывода матрицы смежности на экран(toMatrix) и добавления связей между вершинами(addEdge).



Функции прохода графа в глубину для списка смежности и матрицы смежности между собой очень похожи. Обе функции устроенные на принципе рекурсии, то есть функция вызывает сама себя, передавая новые значения. Алгоритм состоит из подготовительной части, в которой все вершины помечаются как не помеченные и осуществляется запуск процедуры обхода для вершин графа. И непосредственно процедуры обхода, которая помечает текущую (т.е. ту, в которой на текущей итерации находится алгоритм) вершину как посещенную. Затем выводит номер текущей вершины на экран и в цикле просматривает **v**-ю строку матрицы смежности графа G(v,i). Как только алгоритм встречает смежную с **v** не посещенную вершину, то для этой вершины вызывается процедура обхода.



Функция прохода графа в глубину для матрицы смежности, реализованная без рекурсии. Алгоритм очень похож на проход графа в ширину, только во главе стоит не очередь, а стек. Элементы на примере стека добавляются в динамический список и выходя из него, после чего программа проверяет уже выше сказанную проверку и выводит результат.

**Результат работы программы:**

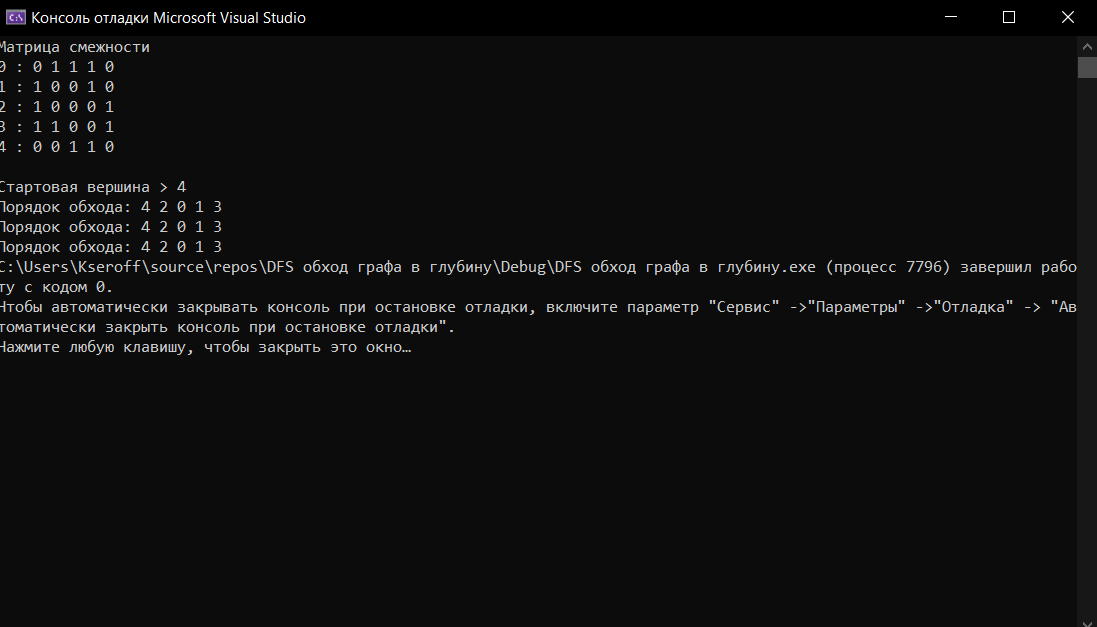


Рисунок 1- результат работы на 5 вершин

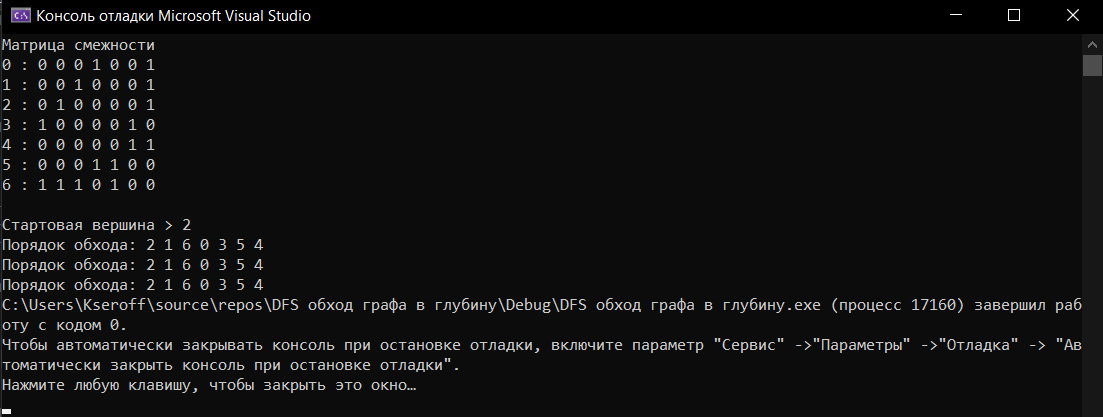


Рисунок 2- результат работы на 7 вершин

**Листинг:**

**DFS**

#include <iostream>

#include <vector>

using namespace std;

template<typename T>

class MyList {

public:

class Iterator;

MyList() : size(0), head(nullptr), tail(nullptr) {}

~MyList() { delete head; delete tail; }

T GetSize() { return size; }

T GetTail() { return tail->data; }

bool isEmpty() { return(head == nullptr); }

void push\_back(T data);

void pop\_back();

Iterator begin() { return Iterator(head); }

Iterator end() { return Iterator(nullptr); }

private:

template<typename T>

struct Node {

T data;

Node\* next;

Node\* pred;

Node(T data = T(), Node\* pNext = nullptr, Node\* pPred = nullptr) {

this->data = data;

this->next = pNext;

this->pred = pPred;

}

};

int size;

Node<T>\* head;

Node<T>\* tail;

};

template<typename T>

class MyList<T>::Iterator

{

Node<T>\* ptr;

public:

Iterator() : ptr(nullptr) {};

Iterator(Node<T>\* p) : ptr(p) {}

Iterator& operator ++ ()

{

if (ptr)

{

ptr = ptr->next; //Может быть еще и здесь кинуть исключение

return \*this;

}

}

Iterator& operator -- ()

{

if (ptr)

{

ptr = ptr->pred; //Может быть еще и здесь кинуть исключение

return \*this;

}

else

{

ptr = tail;

return \*this;

}

}

T& operator \* ()

{

if (ptr)

return ptr->data;

}

bool operator != (const Iterator& it) { return ptr != it.ptr; }

bool operator == (const Iterator& it) { return ptr == it.ptr; }

};

template<typename T>

void MyList<T>::push\_back(T data)

{

Node<T>\* temp = new Node<T>(data);

temp->next = 0;

temp->data = data;

temp->pred = tail;

if (tail != nullptr)

tail->next = temp;

if (size == 0) {

head = tail = temp;

}

else {

tail = temp;

}

size++;

}

template<typename T>

void MyList<T>::pop\_back()

{

Node<T>\* temp = tail;

tail = tail->pred;

delete temp;

size--;

(size == 0) ? head=tail=nullptr : tail->next = nullptr;

}

class Graph {

private:

///////////////////

bool\*\* adjMatrix;

bool\* visited;

bool\* v;

int numVertices;

//////////////////

MyList<int>\* adjLists;

bool\* visit;

public:

Graph(int numVertices)

{

this->numVertices = numVertices;

adjMatrix = new bool\* [numVertices];

visited = new bool[numVertices];

adjLists = new MyList<int>[numVertices];

visit = new bool[numVertices];

v = new bool[numVertices];

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

adjMatrix[i] = new bool[numVertices];

visited[i] = false;

visit[i] = false;

v[i] = false;

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

adjMatrix[i][j] = false;

}

}

~Graph() {

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

delete[] adjMatrix[i];

delete[] adjMatrix;

delete[]visited;

delete[]visit;

delete[]v;

}

void toMatrix() {

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

cout << i << " : ";

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

cout << adjMatrix[i][j] << " ";

}

cout << endl;

}

}

void addEdge(int i, int j) {

adjMatrix[i][j] = adjMatrix[j][i] = true;

adjLists[i].push\_back(j);

adjLists[j].push\_back(i);

}

void DFS\_M(int st);

void DFS(int start);

void DFS\_S(int v);

};

void Graph::DFS\_S(int start)

{

visit[start] = true;

cout << start << " ";

for (auto i = adjLists[start].begin(); i != adjLists[start].end(); ++i)

if (!visit[\*i])

DFS\_S(\*i);

}

void Graph:: DFS\_M(int start)

{

visited[start] = true;

cout << start << " ";

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

if ((adjMatrix[start][i] != 0) && (!visited[i]))

DFS\_M(i);

}

void Graph::DFS(int start) {

MyList<int> S;

int t;

S.push\_back(start);

v[start]++;

while (!S.isEmpty())

{

t = S.GetTail();

S.pop\_back();

cout << t << " ";

for (int i = numVertices - 1; i >= 0; i--)

if (adjMatrix[t][i] && !v[i])

{

S.push\_back(i);

v[i]++;

}

}

}

int main()

{

srand(time(0));

setlocale(0, "");

int z = 5;

Graph c(z);

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

{

for (int j = i + 1; j < z; j++)

{

if (bool(rand() % 2))

c.addEdge(i, j);

}

}

cout << "Матрица смежности" << endl;

c.toMatrix();

int start;

cout<<endl << "Стартовая вершина > "; cin >> start;

cout << "Порядок обхода: ";

c.DFS\_M(start);

cout << endl<< "Порядок обхода: ";;

c.DFS\_S(start);

cout<<endl << "Порядок обхода: ";

c.DFS(start);

cin.get();

return 0;

}

**Вывод:** Изучили основные алгоритмы обхода графа и научились решать задачи обхода графа на основе поиска в глубину.