МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ

ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ОБРАЗОВАНИЯ

«НОВОСИБИРСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ»



**ОТЧЁТ**

**по лабораторной работе № 7-8**

**«Графы*»***

по дисциплине:

**«Программирование»**

|  |  |
| --- | --- |
| Выполнил(а):  Дашкин С.М. Группа АТ-34 | Проверил:  Кухто А.В. |

Новосибирск

2024

**ЦЕЛЬ РАБОТЫ**

1. Разработайте программу, реализующую генерирование графа дружеских отношений и вывод его на экран, считывание данных для построения графа производите из файла (создание элементов графа в динамической памяти, ввод значений вершин графа из файла, установка связей между вершинами).
2. Реализуйте функцию, находящую ответ на вопрос, в соответствии с заданием (на основании построенного графа).
3. Реализуйте функцию удаления вершин графа и ее связей.
4. Реализуйте алгоритм обхода/поиска кратчайшего пути графа.
5. Оформите отчет по лабораторной работе.

|  |  |
| --- | --- |
| **Вариант   6** | |
| **Представление графа** | Список смежности, граф ориентированный, невзвешенный |
| **Найдите ответ на вопрос** | Если я твой друг, то ты мой друг? |
| **Действие** | Реализовать поиск пути между двумя вершинами с минимальным количеством ребер (поиск в глубину) |

**ХОД РАБОТЫ:**

Код реализует структуру дерева, где каждая вершина имеет значение типа char и два указателя на левого и правого потомков.

1. create\_graph(int num\_vertices) - функция создает граф с заданным количеством вершин. Выделяется память для структуры графа и списка смежности. Список смежности инициализируется пустыми списками для каждой вершины.

2. add\_edge(Graph\* graph, int src, int dest, int weight) - добавляет ребро в граф. Она создает новый узел, содержащий вершину назначения и вес ребра, и добавляет его в список смежности исходной вершины.

3. print\_graph(const Graph\* graph) - печатает граф. Она проходит по списку смежности каждой вершины и печатает вершины, с которыми она связана, и вес ребра.

4. deikstra(const Graph\* graph, int start, int end, int\* path, int\* path\_length) - функция реализует алгоритм Дейкстры для поиска кратчайшего пути между двумя вершинами в графе. Она выделяет память для массивов расстояний, предыдущих вершин и посещенных вершин. Затем она выполняет алгоритм Дейкстры и сохраняет кратчайший путь в массиве path и его длину в path\_length.

5. main() - главная функция программы. Она создает граф, добавляет ребра, печатает граф, находит кратчайший путь между двумя вершинами и освобождает память.

**ЛИСТИНГ:**

#define \_CRT\_SECURE\_NO\_DEPRECATE

#include <stdio.h>

#include <stdlib.h>

#include <locale.h>

#include <limits.h>

typedef struct Node {

int vertex;

int weight;

struct Node\* next;

} Node;

typedef struct Graph {

int num\_vertices;

Node\*\* adj\_list;

} Graph;

Graph\* create\_graph(int num\_vertices) {

Graph\* graph = (Graph\*)malloc(sizeof(Graph));

graph->num\_vertices = num\_vertices;

graph->adj\_list = (Node\*\*)malloc(num\_vertices \* sizeof(Node\*));

for (int i = 0; i < num\_vertices; i++) {

graph->adj\_list[i] = NULL;

}

return graph;

}

void add\_edge(Graph\* graph, int src, int dest, int weight) {

Node\* new\_node = (Node\*)malloc(sizeof(Node));

new\_node->vertex = dest;

new\_node->weight = weight;

new\_node->next = graph->adj\_list[src];

graph->adj\_list[src] = new\_node;

}

void print\_graph(const Graph\* graph) {

for (int i = 0; i < graph->num\_vertices; i++) {

printf("Вершина %d: ", i);

Node\* temp = graph->adj\_list[i];

while (temp != NULL) {

printf("%d (%d) -> ", temp->vertex, temp->weight);

temp = temp->next;

}

printf("NULL\n");

}

}

void deikstra(const Graph\* graph, int start, int end, int\* path, int\* path\_length) {

int\* dist = (int\*)malloc(graph->num\_vertices \* sizeof(int));

int\* prev = (int\*)malloc(graph->num\_vertices \* sizeof(int));

int\* visited = (int\*)calloc(graph->num\_vertices, sizeof(int));

for (int i = 0; i < graph->num\_vertices; i++) {

dist[i] = INT\_MAX;

prev[i] = -1;

}

dist[start] = 0;

for (int i = 0; i < graph->num\_vertices - 1; i++) {

int min\_dist = INT\_MAX;

int u = -1;

for (int j = 0; j < graph->num\_vertices; j++) {

if (!visited[j] && dist[j] < min\_dist) {

min\_dist = dist[j];

u = j;

}

}

visited[u] = 1;

Node\* temp = graph->adj\_list[u];

while (temp != NULL) {

int v = temp->vertex;

int weight = temp->weight;

if (!visited[v] && dist[u] + weight < dist[v]) {

dist[v] = dist[u] + weight;

prev[v] = u;

}

temp = temp->next;

}

}

int curr = end;

\*path\_length = 0;

while (curr != -1) {

path[\*path\_length] = curr;

(\*path\_length)++;

curr = prev[curr];

}

free(dist);

free(prev);

free(visited);

}

int main() {

setlocale(LC\_ALL, "Russian");

int num\_vertices = 5;

int num\_edges = 6;

int edges[6][3] = { {0, 1, 1}, {1, 0, 1}, {0, 2, 1}, {2, 0, 1}, {1, 3, 1}, {3, 1, 1} };

Graph\* graph = create\_graph(num\_vertices);

for (int i = 0; i < num\_edges; i++) {

add\_edge(graph, edges[i][0], edges[i][1], edges[i][2]);

}

print\_graph(graph);

int\* path;

int path\_length = 0;

path = (int\*)malloc(num\_vertices \* sizeof(int));

deikstra(graph, 0, 3, path, &path\_length);

printf("Кратчайший путь между 0 и 3: ");

for (int i = path\_length - 1; i >= 0; i--) {

printf("%d ", path[i]);

}

printf("\n");

free(path);

for (int i = 0; i < graph->num\_vertices; i++) {

Node\* current = graph->adj\_list[i];

while (current != NULL) {

Node\* temp = current->next;

free(current);

current = temp;

}

}

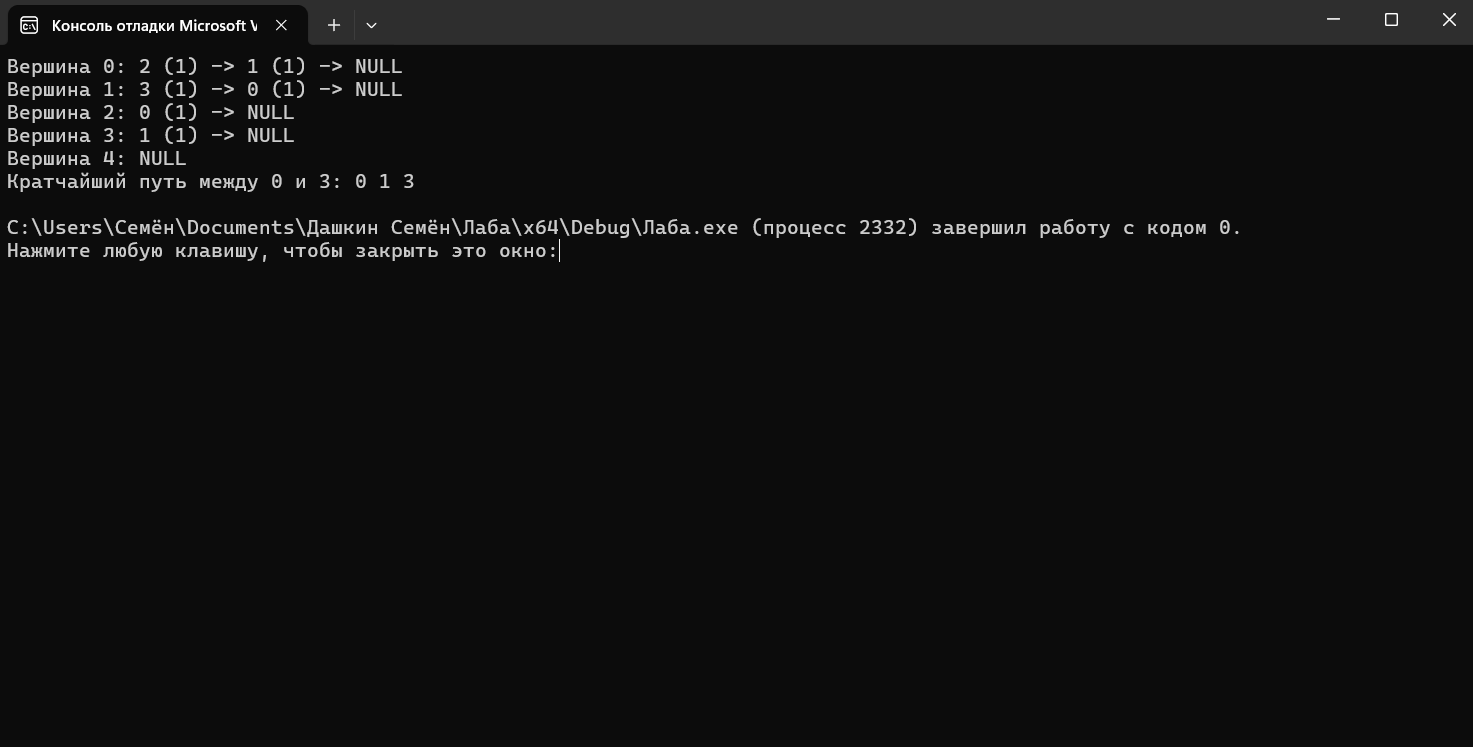
free(graph->adj\_list);

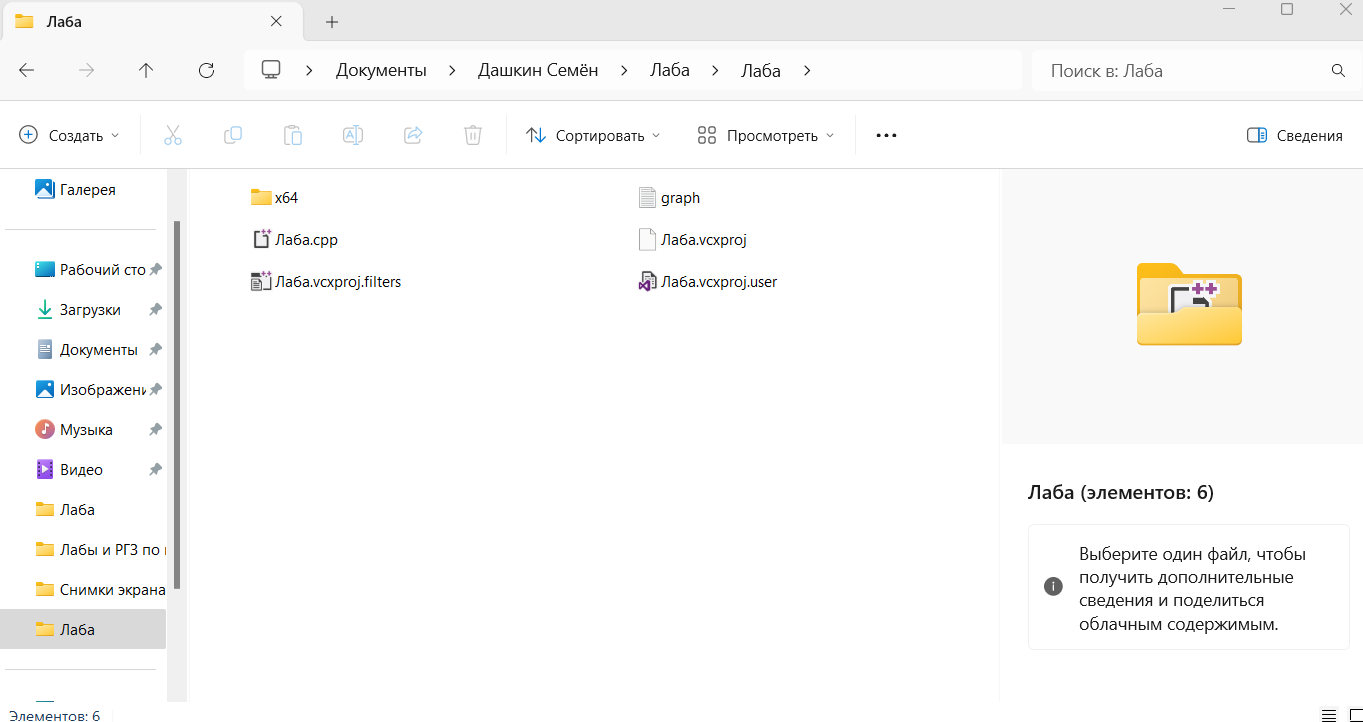
free(graph);

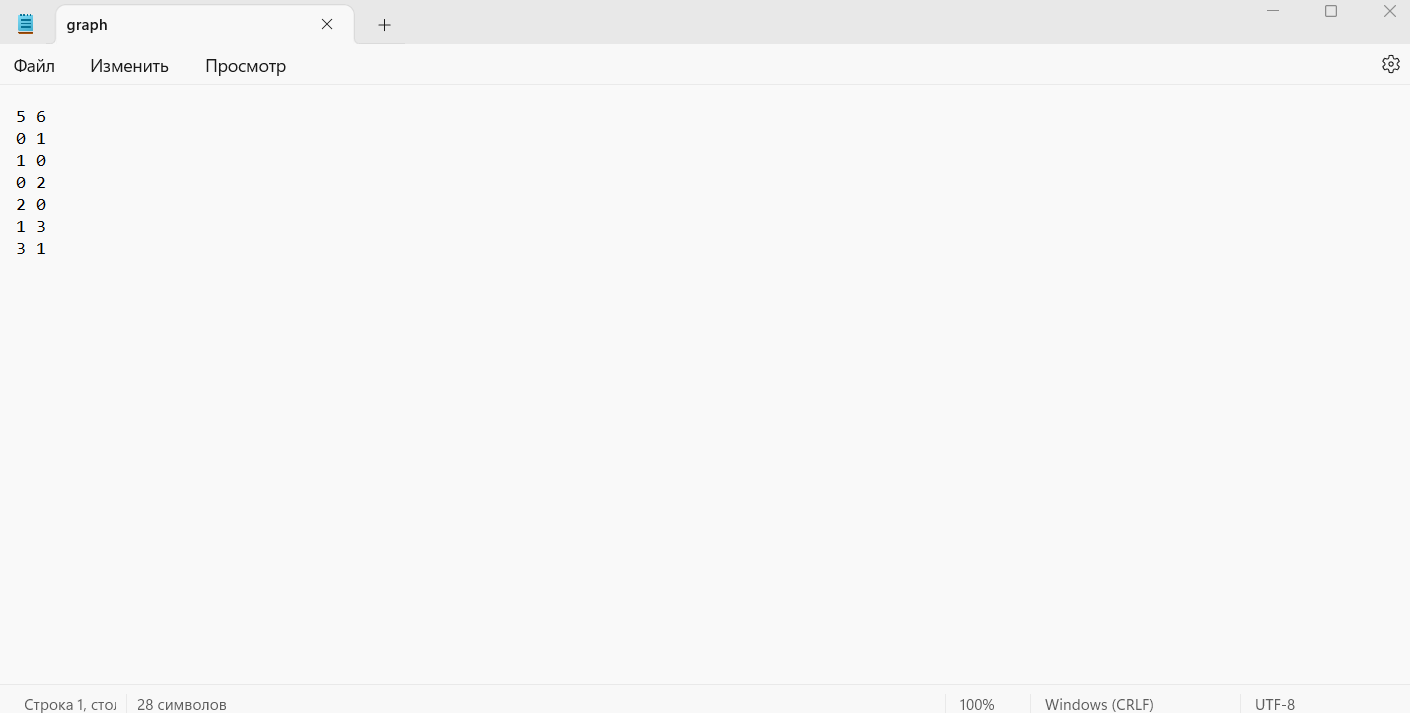
return 0;

}

**ТЕСТИРОВАНИЕ:**

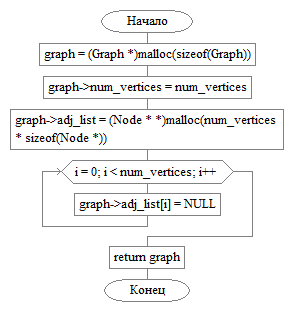


****

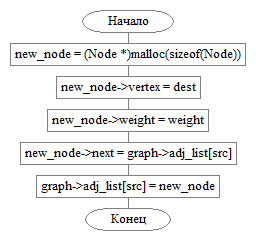
****

**БЛОК СХЕМЫ:**

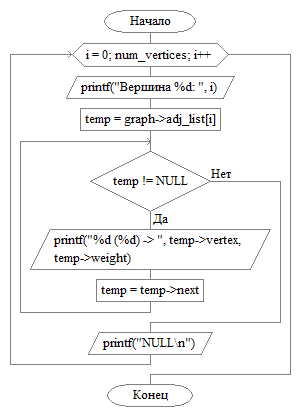
Блок-схема функции create\_graph:



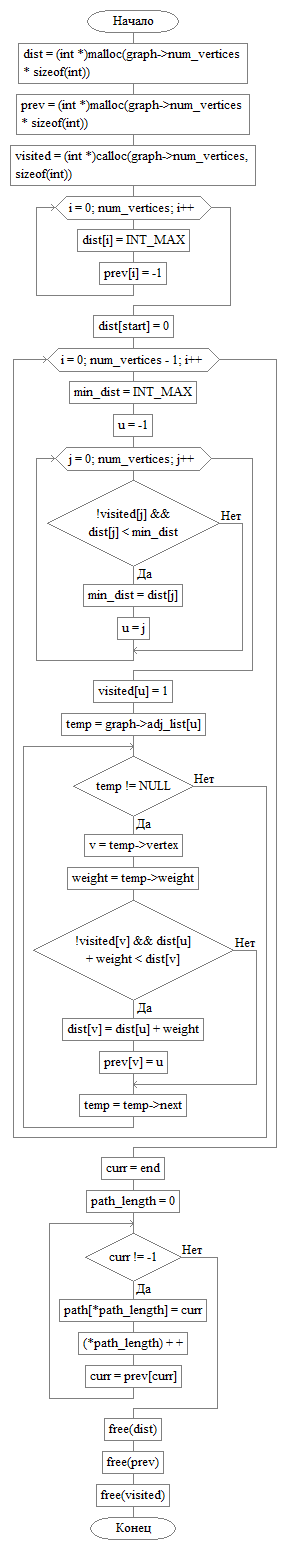
Блок-схема функции add\_edge:



Блок-схема функции print\_graph:



Блок-схема функции deikstra:



Блок-схема функции main:

