Министерство науки и высшего образования Российской Федерации  
Пензенский государственный университет  
Кафедра вычислительная техника

**ОТЧËТ**  
по лабораторной работе №7  
по дисциплине: «Обход графа в глубину»

Выполнили студенты группы 22ВВВ2:  
Беляев Д.

Приняли:  
Акифьев И. В.

Митрохин М. А.

Пенза 2023

**Название**

Обход графа в глубину

**Цель работы**

Разработать и реализовать алгоритм обхода графа в глубину

**Лабораторное задание**

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.

**Листинг**

#include <stdio.h>

#include <time.h>

#include <stdlib.h>

#include "list.h"

int\*\* GenerateMatrix(int size\_x, int size\_y);

void ClearMatrix(int\*\* matrix, int size);

void PrintMatrix(int\*\* matrix, int size\_x, int size\_y);

void PrintListMatrix(DataType\*\* matrix, int size);

int\*\* GenerateAdjacencyMatrix(int size);

DataType\*\* GenerateListMatrix(int\*\* matrix, int size);

void DFSadj(int\*\* matrix, int matrixSize);

void DFSlist(DataType\*\* list, int listSize);

void DFSNonRecursion(int\*\* matrix, int size);

int main()

{

srand(time(NULL));

const int size = 9;

int\*\* matrix = GenerateAdjacencyMatrix(size);

printf("Adjacency matrix:\n");

PrintMatrix(matrix, size, size);

printf("---- START DEPTH RUN ----\n");

DFSadj(matrix, size);

printf("\n--------\n\n");

DataType\*\* list = GenerateListMatrix(matrix, size);

printf("List matrix:\n");

PrintListMatrix(list, size);

printf("---- START DEPTH RUN ----\n");

DFSlist(list, size);

printf("\n--------\n\n");

printf("---- START NON RECURSION DEPTH RUN ----\n");

DFSNonRecursion(matrix, size);

return 0;

}

#pragma region Matrix work

int\*\* GenerateMatrix(int size\_x, int size\_y)

{

int\*\* matrix = (int\*\*)malloc(size\_x \* sizeof(int\*));

if (matrix == NULL)

return NULL;

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

{

matrix[i] = (int\*)malloc(size\_y \* sizeof(int));

if (matrix[i] == NULL)

return NULL;

}

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

{

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

{

matrix[i][j] = 0;

}

}

return matrix;

}

void ClearMatrix(int\*\* matrix, int size)

{

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

{

free(matrix[i]);

}

free(matrix);

}

void PrintMatrix(int\*\* matrix, int size\_x, int size\_y)

{

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

{

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

{

printf("%d ", matrix[i][j]);

}

printf("\n");

}

}

void PrintListMatrix(DataType\*\* matrix, int size)

{

for (size\_t i = 0; i < size; i++)

{

PrintDataType(matrix[i]);

}

}

int\*\* RemoveVertex(int\*\* matrix, int\* size, int r)

{

int s = \*size;

int\*\* nMatrix = GenerateMatrix(s - 1, s - 1);

int di = 0, dj = 0;

for (size\_t i = 0; i < s; i++)

{

if (i == r)

i++;

if (i >= s)

break;

for (size\_t j = 0; j < s; j++)

{

if (j == r)

j++;

if (j >= s)

break;

nMatrix[di][dj] = matrix[i][j];

dj++;

}

di++;

dj = 0;

}

\*size = s - 1;

return nMatrix;

}

#pragma endregion

#pragma region MatrixGenerate

int\*\* GenerateAdjacencyMatrix(int size)

{

int\*\* matrix = GenerateMatrix(size, size);

if (matrix == NULL)

return NULL;

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

{

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

{

if (i == j)

continue;

int rnd = rand() % 2;

matrix[i][j] = rnd;

if (rnd == 1)

matrix[j][i] = 1;

}

}

return matrix;

}

DataType\*\* GenerateListMatrix(int\*\* matrix, int size)

{

DataType\*\* matrixL = (DataType\*\*)malloc(sizeof(DataType\*) \* size);

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

matrixL[i] = CreateDataType(i);

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

{

for (size\_t j = 0; j < size; j++)

{

if (matrix[i][j] != 1)

continue;

Add(j, matrixL[i]);

}

}

return matrixL;

}

#pragma endregion

#pragma region FirstNumber

void DFSCuroadj(int start, int\* visited, int\*\* matrix, int matrixSize)

{

visited[start] = 1;

printf("%d ", start);

for (int r = 0; r <= matrixSize; r++)

{

if ((matrix[start][r] != 0) && (visited[r] == 0))

{

DFSCuroadj(r, visited, matrix, matrixSize);

}

}

}

void DFSadj(int\*\* matrix, int matrixSize)

{

int\* vis = (int\*)calloc(matrixSize, sizeof(int));

DFSCuroadj(0, vis, matrix, matrixSize);

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

{

if (vis[i] == 0)

{

printf("\n");

DFSCuroadj(i, vis, matrix, matrixSize);

}

}

printf("\n");

free(vis);

}

void DFSCurolist(int start, int\* visited, DataType\*\* list, int listSize)

{

visited[start] = 1;

printf("%d ", start);

int i = 1;

int\* v = TryCheckElementAt(i, list[start]);

while (v != NULL)

{

if (v == NULL)

break;

if (visited[\*v] == 0)

{

DFSCurolist(\*v, visited, list, listSize);

}

i++;

v = TryCheckElementAt(i, list[start]);

}

}

void DFSlist(DataType\*\* list, int listSize)

{

int\* vis = (int\*)calloc(listSize, sizeof(int));

DFSCurolist(0, vis, list, listSize);

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

{

if (vis[i] == 0)

{

printf("\n");

DFSCurolist(i, vis, list, listSize);

}

}

printf("\n");

free(vis);

}

#pragma endregion

#pragma region SecondNumber

typedef struct Stack

{

int maxSize;

int\* data;

size\_t size;

} Stack;

Stack\* CreateStack(int size)

{

Stack\* tmp = (Stack\*)malloc(sizeof(Stack));

tmp -> data = (int\*)malloc(sizeof(int) \* size);

tmp -> maxSize = size;

tmp -> size = 0;

return tmp;

}

void Push(Stack\* stack, int val)

{

if (stack->size >= stack->maxSize) {

return;

}

stack->data[stack->size] = val;

stack->size++;

}

int\* Pop(Stack\* stack)

{

if (stack->size <= 0) {

return NULL;

}

stack->size--;

return &(stack->data[stack->size]);

}

int IsStackEmpty(Stack\* stack)

{

if (stack->size <= 0)

return 0;

return 1;

}

void ClearStack(Stack\* stack)

{

free(stack->data);

free(stack);

}

void DFSNonRecursionLogic(int\*\* matrix, int size, int vertex, int\* visited)

{

Stack\* st = CreateStack(INT\_MAX);

Push(st, vertex);

while (IsStackEmpty(st) == 1)

{

int\* vert = Pop(st);

if (vert == NULL)

return;

vertex = \*vert;

if (visited[vertex] == 1)

continue;

visited[vertex] = 1;

printf("%d ", vertex);

for (size\_t i = 0; i < size; i++)

{

if (matrix[vertex][i] == 1 && visited[i] == 0)

Push(st, i);

}

}

ClearStack(st);

}

void DFSNonRecursion(int\*\* matrix, int size)

{

int\* visited = (int\*)calloc(size, sizeof(int));

DFSNonRecursionLogic(matrix, size, 0, visited);

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

{

if (visited[i] == 0)

{

printf("\n");

DFSNonRecursionLogic(matrix, size, i, visited);

}

}

printf("\n");

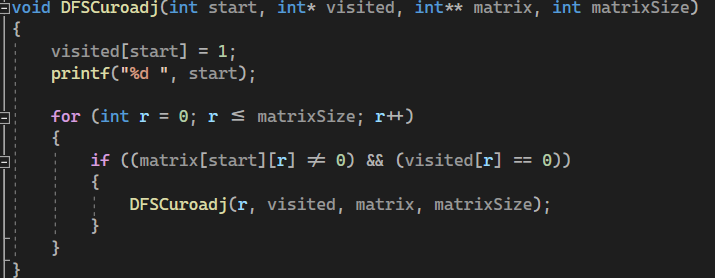
free(visited);

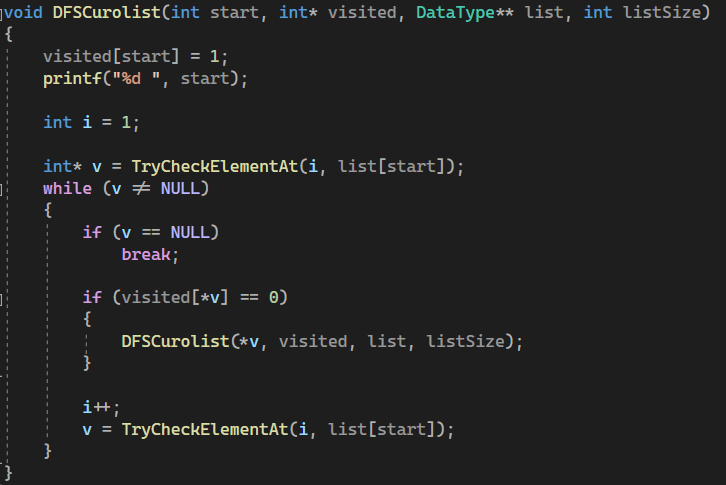
}

#pragma endregion

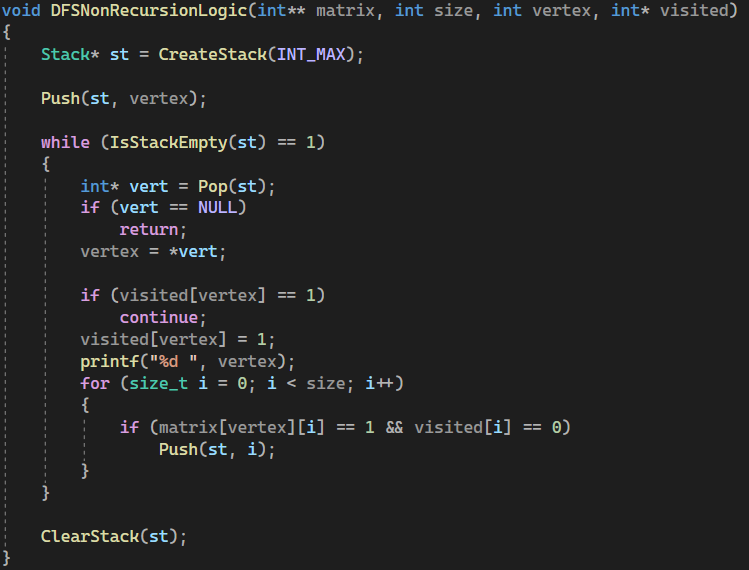
**Задания**

Задание 1

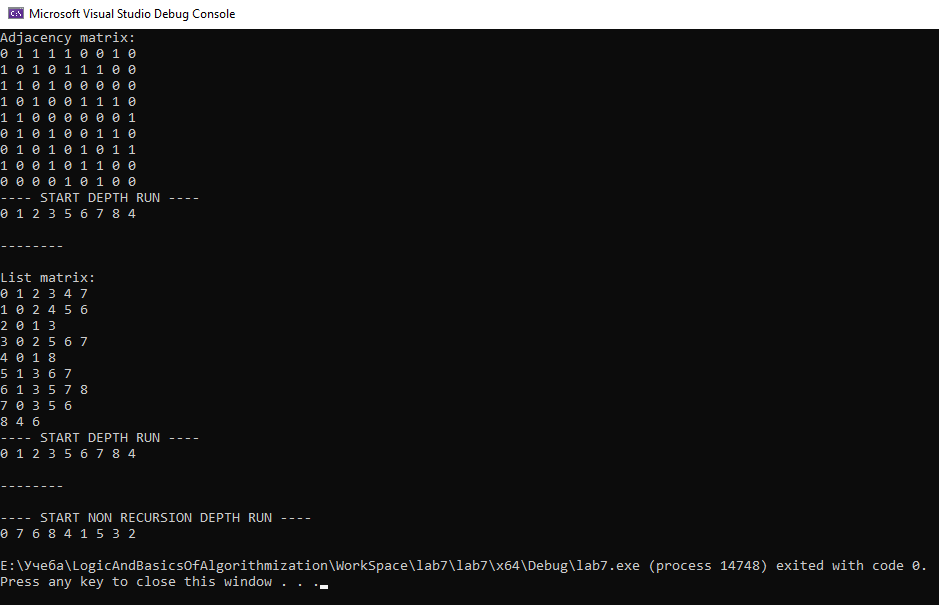




Задание 2



**Результат работы программы**

****

**Вывод**

Разработал и реализовал алгоритм обхода графа в глубину