МИНИСТЕРВСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

Кафедра «Вычислительная техника»

ОТЧЕТ

по лабораторной работе №8

по дисциплине: "Логика и основы алгоритмизации в инженерных задачах"

на тему: "Обход графа в ширину"

Выполнили студенты группы 24ВВВ3:

Пяткин Р. С.

Гусаров Е. Е.

Принял:

к.т.н., доцент, Юрова О. В.

к.т.н., Деев М. В.

Пенза 2025

Цель

Изучение обхода графа в ширину.

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 ширину, реализованную в соответствии с приведенным выше описанием. При реализации алгоритма в качестве очереди используйте класс **queue** из стандартной библиотеки C++.

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

Задание 2 *

- Для матричной формы представления графов реализуйте алгоритм обхода в ширину с использованием очереди, построенной на основе структуры данных «список», самостоятельно созданной в лабораторной работе № 3.

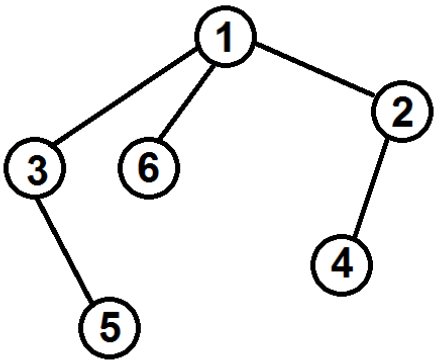
- Оцените время работы двух реализаций алгоритмов обхода в ширину (использующего стандартный класс **queue** и использующего очередь, реализованную самостоятельно) для графов разных порядков.

Теория

Обход графа в ширину – еще один распространенный способ обхода графов.

Основная идея такого обхода состоит в том, чтобы посещать вершины по уровням удаленности от исходной вершины. Удалённость в данном

случае понимается как количество ребер, по которым необходимо прейти до достижения вершины. Например, если для графа на рисунке 1 начать обход из первой вершины, то вершины 3, 6 и 2 будут находиться на уровне удаленности в 1 ребро, а вершины 5 и 4 на уровне удаленности в 2 ребра.



$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

Рисунок 1 – Граф

Тогда при обходе этого графа в ширину, мы сначала посетим вершины первого уровня удаленности (с номерами 2, 3 и 6), и только после того, как закончатся не посещенные вершины на этом уровне, мы перейдем к следующему. На втором уровне мы посетим все вершины, которые удалены от исходной на 2 ребра (вершины 4 и 5).

Так, алгоритм обхода в ширину продолжает осматривать уровень за уровнем, пока не пройдет все доступные вершины.

Чтобы не заходить повторно в уже пройденные вершины, они помечаются, как и в алгоритме обхода в глубину.

Для того, чтобы проход осуществлялся по уровням необходимо хранить информацию о требуемом порядке посещения вершин. Вершины, которые являются ближайшими соседями исходной вершины (из которой начат обход) должны быть посещены раньше, чем соседи соседей и т.д. Такой порядок позволяет задать структура данных «очередь». Просматривая строку матрицы смежности (или список смежности) для текущей вершины мы помещаем всех её ещё не посещенных соседей в очередь. На следующей

итерации текущей вершиной становится та, которая стоит в очереди первой и уже её не посещенные соседи будут помещены в очередь. Но место в очереди они займут после тех вершин, которые были помещены туда на предыдущих итерациях.

Таким образом, можно предложить следующую реализацию алгоритма обхода в ширину.

Реализация

Вход: G – матрица смежности графа.

Выход: номера вершин в порядке их прохождения на экране.

Алгоритм ПОШ

1.1. для всех i положим NUM[i] = False пометим как "не посещенную";

1.2. ПОКА существует "новая" вершина v

1.3. ВЫПОЛНЯТЬ BFS (v).

Алгоритм BFS(v):

2.1. Создать пустую очередь Q = {};

2.2. Поместить v в очередь Q.push(v);

2.3. пометить v как "посещенную" NUM[v] = True;

2.4. ПОКА Q != Æ очередь не пуста ВЫПОЛНЯТЬ

2.5. v = Q.front() установить текущую вершину;

2.6. Удалить первый элемент из очереди Q.pop();

2.7. вывести на экран v;

2.8. ДЛЯ i = 1 ДО size_G ВЫПОЛНЯТЬ

2.9. ЕСЛИ G(v,i) = = 1И NUM[i] = = False

2.10. ТО

2.11. Поместить i в очередь Q.push(i);

2.12. пометить v как "посещенную" NUM[v] = True;

Реализация состоит из подготовительной части, в которой все вершины помечаются как не посещенные (п.1.1) и осуществляется запуск процедуры обхода для вершин графа (п.1.2, 1.3).

В самой процедуре обхода сначала создается пустая очередь (п. 2.1), в которую помещается исходная вершина, из которой начат обход (п.2.2).

Далее итерационно, пока очередь не опустеет, из нее извлекается первый элемент, который становится текущей вершиной (п. 2.5, 2.6). Затем в цикле просматривается v-я строка матрицы смежности графа G(v,i). Как только алгоритм встречает смежную с v не посещенную вершину (п.2.9), эта вершина помещается в очередь (п.2.11) и помечается как посещенная (п.2.12). После просмотра строки матрицы смежности алгоритм делает следующую итерацию цикла 2.4 или заканчивает работу, если очередь пуста.

Так, если для графа на рисунке 1, мы начнем обход из первой вершины, то на шаге 2.2 она будет помещена в очередь и помечена как посещенная (NUM[1] = True). Условие цикла 2.4 будет выполнено, вершина 1 будет установлена в качестве текущей (п.2.5) и удалена из очереди (п.2.6). На экран будет выведена единица.

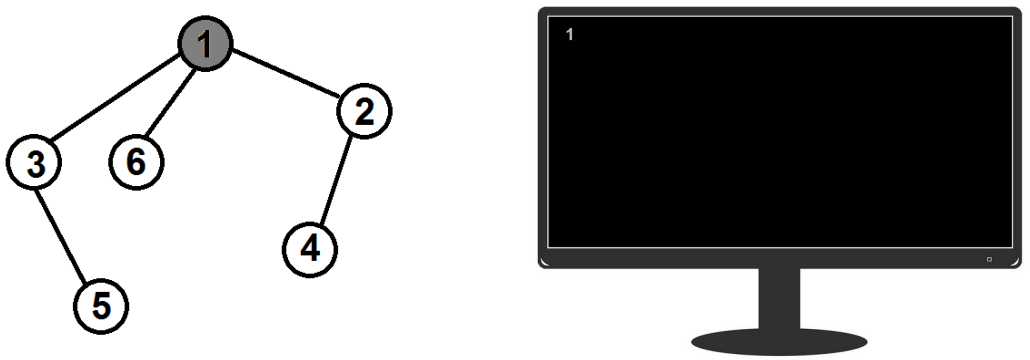


Рисунок 2 – Итерация 1

При просмотре 1-й строки матрицы смежности

$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

будет найдена смежная вершина с индексом 2 (G(1,2) = = 1), которая не посещена (NUM[2] = = False) и она будет помещена в очередь и помечена (NUM[2] = = True). Просмотр строки продолжится, и в очередь будут также помещены и помечены как посещенные вершины с индексами 3 и 6.

К концу первой итерации очередь будет содержать Q = {2, 3, 6}.

Условие цикла while будет выполнено и на следующей итерации вершина 2 будет установлена как текущая и извлечена из очереди, на экран будет выведена двойка.

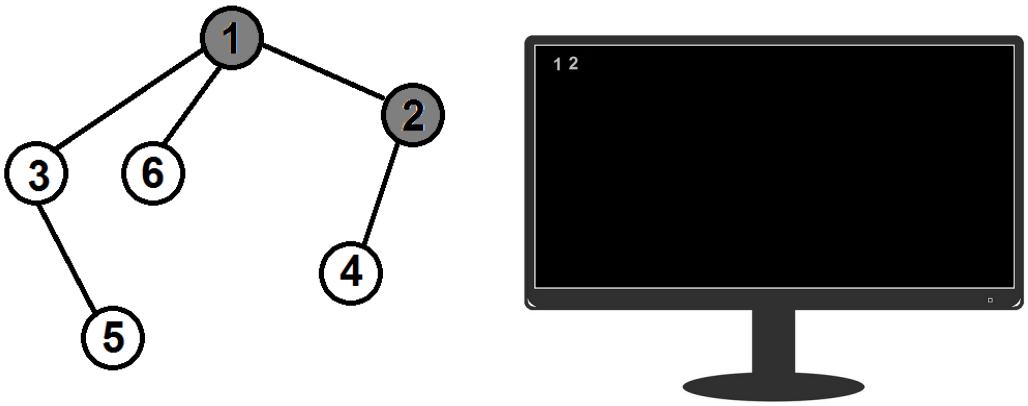


Рисунок 3 – Итерация 2

И алгоритм перейдет к просмотру второй строки матрицы смежности. Первая смежная с вершиной 2 - вершина с индексом 1(G(2,1) = =1),

$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

которая к настоящему моменту уже посещена (NUM[1] = = True), она не будет помещена в очередь. Цикл 2.9 продолжит просмотр матрицы смежности.

$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

Следующая найденная вершина, смежная со второй, будет иметь индекс 4 (G(2,4) = =1), она не посещена (NUM[4] = = False) и она будет помещена в очередь и помечена.

К концу второй итерации очередь будет содержать Q = {3, 6, 4}.

Условие цикла while будет выполнено и на следующей итерации вершина 3 будет установлена как текущая и извлечена из очереди, на экран будет выведена тройка.

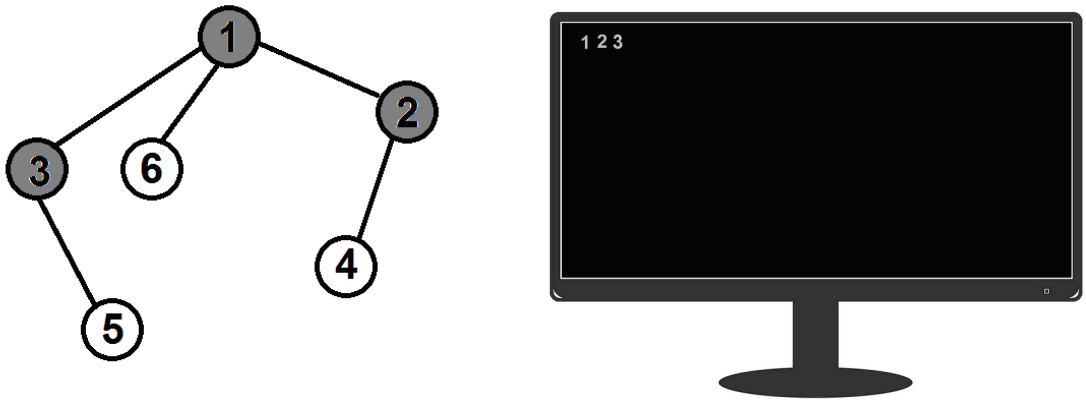


Рисунок 4 – Итерация 3

При просмотре 3-й строки матрицы будет найдена вершина 1, но она уже посещена (NUM[1] = True), поэтому в очередь помещена не будет.

$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

Следующая найденная вершина, смежная с третьей – вершина с номером 5 (G(3,5) = =1), она не посещена (NUM[5] = = False) и будет помещена в очередь.

$$G = \begin{Bmatrix} 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 \end{Bmatrix}$$

К концу третьей итерации очередь будет содержать Q = {6, 4, 5}.

Условие цикла while будет выполнено. На следующей итерации вершина 6 будет установлена как текущая и извлечена из очереди, на экран будет выведена цифра шесть.

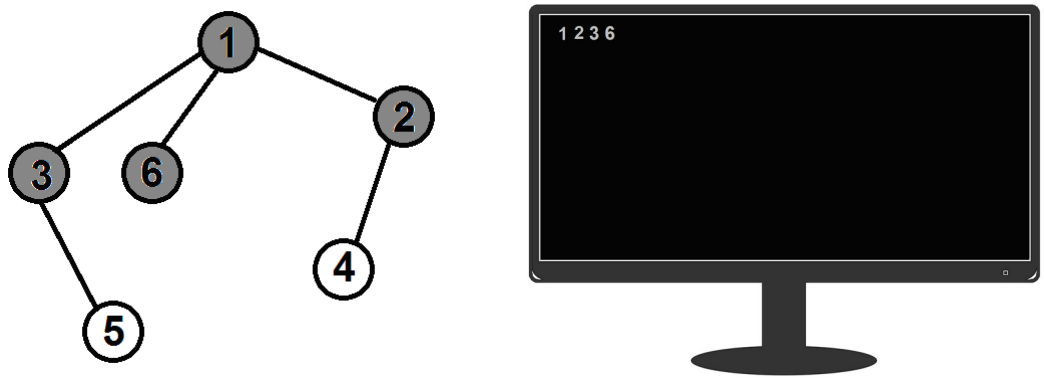


Рисунок 5 – Итерация 4

При просмотре 6-й строки матрицы будет найдена вершина 1, но она уже посещена (NUM[1] = True), поэтому в очередь помещена не будет.

К концу четвертой итерации очередь будет содержать Q = {4, 5}.

Далее из очереди будет извлечена вершина 4, установлена как текущая и выведена на экран.

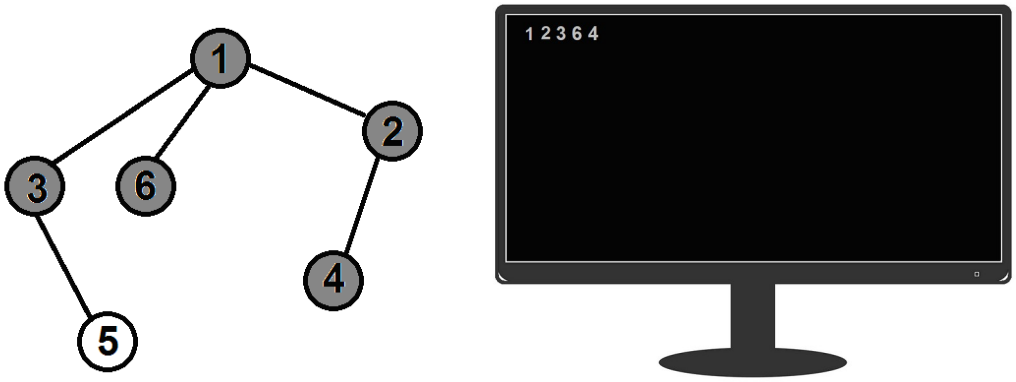


Рисунок 6 – Итерация 5

Так как все вершины кроме пятой уже посещены, то при просмотре 4-й строки в матрице смежности в очередь не будут добавлены вершины.

К концу пятой итерации очередь будет содержать только одну вершину Q = {5}.

На шестой итерации вершина с номером пять будет установлена как текущая и извлечена из очереди, на экран будет выведена цифра пять.

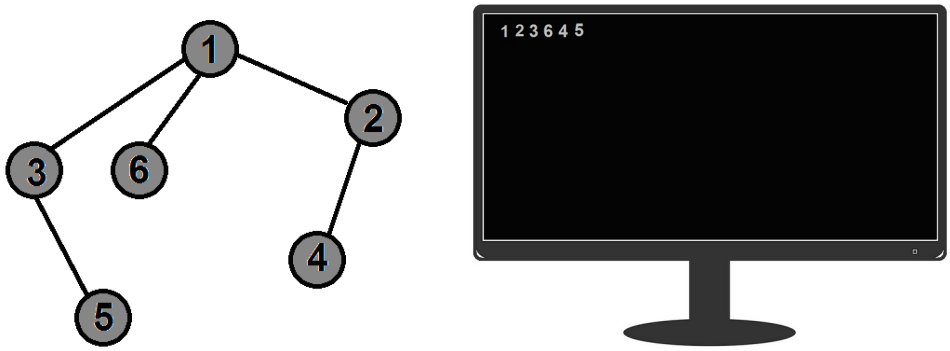


Рисунок 7 – Результат работы обхода

Просмотр 5-й строки матрицы смежности не добавит в очередь новых вершин, так как к этому моменту они уже все помечены как посещенные.

После этой итерации очередь окажется пуста Q = {}и алгоритм завершит свою работу.

В конце работы алгоритма все вершины будут посещены. А на экран будут выведены номера вершин в порядке их посещения алгоритмом.

Вывод:

В ходе выполнения лабораторной работы были разработаны программа для выполнения заданий Лабораторной работы №8. В процессе выполнения работы был изучен способ обхода графа в ширину.

Листинг

Файл lab_8.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <queue>

using namespace std;

int printMatrix(int** m, int n){
cout << " ";
for(int i = 0; i < n; i++){
cout << i << " ";
}
cout << endl;
for(int i = 0; i < n; i++){
cout << i << "| ";
for(int j = 0; j < n; j++){
cout << m[i][j] << " ";
}
cout << endl;
}
cout << endl;
return 0;
}
int** createMatrix(int n){
int** m = new int*[n];
for (int i = 0; i < n; i++) {
m[i] = new int[n];
}

for(int i = 0; i < n; i++){
for(int j = i; j < n; j++){
m[i][j] = m[j][i] = (i == j ? 0 : rand() % 2);
}
}
return m;
}
void deleteM(int** m, int n, int* v) {
for (int i = 0; i < n; ++i) {
delete[] m[i];
}
```

```
delete[] m;
delete[] v;
}

void BFS(int** G, int numG, int* visited, int s) {
queue<int> q;
int v = 0;
visited[s] = 1;
q.push(s);

while (!q.empty())
{
v = q.front();
q.pop();
cout<<v;
for(int i = 0; i < numG; i++){
if(G[v][i] == 1 && visited[i] == 0){
q.push(i);
visited[i] = 1;
}
}
}
}

int main() {
srand(time(0));
int numG = 0;
int n = 0;
cout << "Введите количество вершин в графе: ";
cin >> numG;
if (numG <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
return 1;
}
int* visited = new int[numG];
for(int i = 0; i < numG; i++){
visited[i] = 0;
}
int** G = createMatrix(numG);

cout << "Матрица смежности графа:" << endl;
printMatrix(G, numG);
cout << "Введите начальную вершину обхода: ";
cin >> n;
cout << "Порядок обхода графа в ширину (матрица смежности): ";
BFS(G, numG, visited, n);
cout << endl;
deleteM(G, numG, visited);
return 0;
}
```

Файл lab_8_2.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <queue>

using namespace std;

struct ListNode {
int vertex;
ListNode* next;
ListNode(int v) : vertex(v), next(nullptr) {}
};

void printAdjacencyList(ListNode** adjList, int n) {
cout << "Список смежности графа:" << endl;
for (int i = 0; i < n; i++) {
cout << i << ": ";
ListNode* current = adjList[i];
while (current != nullptr) {
cout << current->vertex << " ";
current = current->next;
}
cout << endl;
}
cout << endl;
}

ListNode** createAdjacencyList(int n) {
ListNode** adjList = new ListNode*[n];
for (int i = 0; i < n; i++) {
adjList[i] = nullptr;
}
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (rand() % 2 == 1) {
ListNode* newNode = new ListNode(j);
newNode->next = adjList[i];
adjList[i] = newNode;
newNode = new ListNode(i);
newNode->next = adjList[j];
adjList[j] = newNode;
}
}
}
return adjList;
}

void deleteAdjacencyList(ListNode** adjList, int n, int* visited) {
for (int i = 0; i < n; i++) {
```

```
ListNode* current = adjList[i];
while (current != nullptr) {
ListNode* temp = current;
current = current->next;
delete temp;
}
}
delete[] adjList;
delete[] visited;
}

void BFS(ListNode** adjList, int numG, int* visited, int start) {
queue<int> q;
visited[start] = 1;
q.push(start);
cout << "Порядок обхода графа в ширину: ";
while (!q.empty()) {
int current = q.front();
q.pop();
cout << current << " ";
ListNode* neighbor = adjList[current];
while (neighbor != nullptr) {
int neighborVertex = neighbor->vertex;
if (visited[neighborVertex] == 0) {
visited[neighborVertex] = 1;
q.push(neighborVertex);
}
neighbor = neighbor->next;
}
}
cout << endl;
}

int main() {
srand(time(0));
int numG = 0;
int startVertex = 0;
cout << "Введите количество вершин в графе: ";
cin >> numG;
if (numG <= 0) {
cout << "Ошибка: количество вершин должно быть положительным!" << endl;
return 1;
}
int* visited = new int[numG];
for (int i = 0; i < numG; i++) {
visited[i] = 0;
}
ListNode** adjList = createAdjacencyList(numG);
printAdjacencyList(adjList, numG);
cout << "Введите начальную вершину обхода: ";
cin >> startVertex;
```

```
if (startVertex < 0 || startVertex >= numG) {
cout << "Ошибка: начальная вершина должна быть в диапазоне [0, " << numG - 1 << "]!" << endl;
deleteAdjacencyList(adjList, numG, visited);
return 1;
}
BFS(adjList, numG, visited, startVertex);
deleteAdjacencyList(adjList, numG, visited);
return 0;
}
```

Файл lab_8_3.cpp

```
#include <iostream>
#include <ctime>
#include <random>
#include <queue>
#include <chrono>
#include <iomanip>

using namespace std;
using namespace std::chrono;

struct ListNode {
int vertex;
ListNode* next;
ListNode(int v) : vertex(v), next(nullptr) {}
};

class Queue {
private:
struct QueueNode {
int data;
QueueNode* next;
QueueNode(int value) : data(value), next(nullptr) {}
};
QueueNode* front;
QueueNode* rear;
public:
Queue() : front(nullptr), rear(nullptr) {}
~Queue() {
while (!isEmpty()) {
pop();
}
}
bool isEmpty() const {
return front == nullptr;
}
void push(int value) {
QueueNode* newNode = new QueueNode(value);
if (rear == nullptr) {
front = rear = newNode;
} else {
```

```
rear->next = newNode;
rear = newNode;
}
}
int pop() {
if (isEmpty()) {
return -1;
}
QueueNode* temp = front;
int value = temp->data;
front = front->next;
if (front == nullptr) {
rear = nullptr;
}
delete temp;
return value;
}
};

ListNode** createAdjacencyList(int n) {
ListNode** adjList = new ListNode*[n];
for (int i = 0; i < n; i++) {
adjList[i] = nullptr;
}
for (int i = 0; i < n; i++) {
for (int j = i + 1; j < n; j++) {
if (rand() % 100 < 1) {
ListNode* newNode = new ListNode(j);
newNode->next = adjList[i];
adjList[i] = newNode;
newNode = new ListNode(i);
newNode->next = adjList[j];
adjList[j] = newNode;
}
}
}
return adjList;
}

void deleteAdjacencyList(ListNode** adjList, int n) {
for (int i = 0; i < n; i++) {
ListNode* current = adjList[i];
while (current != nullptr) {
ListNode* temp = current;
current = current->next;
delete temp;
}
}
delete[] adjList;
}
```

```
void BFS(ListNode** adjList, int numG, int* visited, int start) {
Queue q;
visited[start] = 1;
q.push(start);
while (!q.isEmpty()) {
int current = q.pop();
ListNode* neighbor = adjList[current];
while (neighbor != nullptr) {
int neighborVertex = neighbor->vertex;
if (visited[neighborVertex] == 0) {
visited[neighborVertex] = 1;
q.push(neighborVertex);
}
neighbor = neighbor->next;
}
}
}

void BFS_std(ListNode** adjList, int numG, int* visited, int start) {
queue<int> q;
visited[start] = 1;
q.push(start);
while (!q.empty()) {
int current = q.front();
q.pop();
ListNode* neighbor = adjList[current];
while (neighbor != nullptr) {
int neighborVertex = neighbor->vertex;
if (visited[neighborVertex] == 0) {
visited[neighborVertex] = 1;
q.push(neighborVertex);
}
neighbor = neighbor->next;
}
}
}

int main() {
srand(time(0));
int graphSizes[] = {100, 500, 1000, 2000, 5000, 10000, 30000, 50000};
int numTests = sizeof(graphSizes) / sizeof(graphSizes[0]);
int startVertex = 0;
// Заголовок таблицы
cout << "Сравнение производительности BFS с queue и оптимизированным List:\n";
cout << "Размер графа | queue (мкс) | List (мкс) | Отношение (List/queue)\n";
cout << "-------------|-------------|-------------|----------------------\n";
for (int testIdx = 0; testIdx < numTests; testIdx++) {
int numG = graphSizes[testIdx];
ListNode** adjList = createAdjacencyList(numG);
int* visited = new int[numG];
for (int i = 0; i < numG; i++) visited[i] = 0;
```

```
// Измеряем время для стандартной очереди
auto start = high_resolution_clock::now();
BFS_std(adjList, numG, visited, startVertex);
auto end = high_resolution_clock::now();
auto duration_std = duration_cast<microseconds>(end - start);
// Сбрасываем visited
for (int i = 0; i < numG; i++) visited[i] = 0;
// Измеряем время для самописной очереди
start = high_resolution_clock::now();
BFS(adjList, numG, visited, startVertex);
end = high_resolution_clock::now();
auto duration_custom = duration_cast<microseconds>(end - start);
// Вычисляем отношение
double ratio = (double)duration_custom.count() / duration_std.count();
// Выводим строку таблицы
cout << setw(12) << numG << " | "
<< setw(11) << duration_std.count() << " | "
<< setw(11) << duration_custom.count() << " | "
<< setw(10) << fixed << setprecision(6) << ratio << endl;
delete[] visited;
deleteAdjacencyList(adjList, numG);
}
return 0;
}
```