МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
Кафедра «Вычислительная техника»

ОТЧЁТ

По лабораторной работе №8

По курсу «Логика и основы алгоритмизации в инженерных задачах»

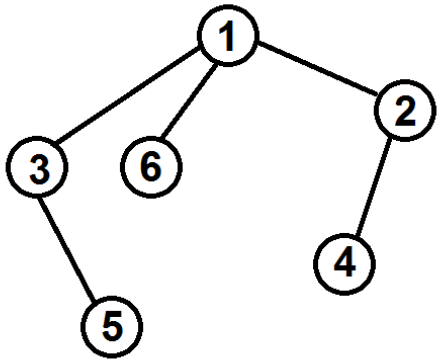
На тему «Обход графа в ширину»

Выполнили
студенты
группы
23ВВВ4:
Святов И.Ю.
Епинин Д.В.

Приняли:
Юрова О.В.
Деев М.В.

Пенза 2024

Общие сведения:
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.

Задание:

1. Сгенерируйте (используя генератор случайных чисел) матрицу смежности для неориентированного графа *G*. Выведите матрицу на экран.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));
    const int G = 5;  // Размер графа
    int M[G][G];      // Матрица смежности

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;  // Нет петли
            }
            else {
                M[i][j] = 0;  // Изначально нет ребер
            }
        }
    }
    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;  // Случайное ребро между i и j
        }
    }
    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

2. Для сгенерированного графа осуществите процедуру обхода в ширину, реализованную в соответствии с приведенным выше описанием.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
```

```
#include <locale.h>

#define G 5  // Размер графа
int M[G][G];      // Матрица смежности
bool visited[G];  // Массив для отслеживания посещенных вершин

void bfs(int start) {
    int q[G];
    int front = 0, rear = 0;
    visited[start] = true;
    q[rear++] = start;

    while (front != rear) {
        int current = q[front++];
        printf("%d ", current);

        for (int i = 0; i < G; i++) {
            if (M[current][i] == 1 && !visited[i]) { // если есть ребро и вершина не
посещена
                visited[i] = true;
                q[rear++] = i;
            }
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;  // Нет петли
            }
            else {
                M[i][j] = 0;  // Изначально нет ребер
            }
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;  // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Обход в ширину, начиная с вершины 0
    printf("Обход в ширину: ");
    bfs(0);
    printf("\n");

    return 0;
}
```

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

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

#define G 5  // Размер графа
// Список смежности
struct Node {
    int vertex;
    struct Node* next;
};

struct Graph {
    int numVertices;
    struct Node** adjList;
};

// Создание нового узла списка смежности
struct Node* createNode(int vertex) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->vertex = vertex;
    newNode->next = NULL;
    return newNode;
}

// Создание графа с заданным количеством вершин
struct Graph* createGraph(int numVertices) {
    struct Graph* graph = (struct Graph*)malloc(sizeof(struct Graph));
    graph->numVertices = numVertices;
    graph->adjList = (struct Node**)malloc(numVertices * sizeof(struct Node*));
    for (int i = 0; i < numVertices; i++) {
        graph->adjList[i] = NULL;
    }
    return graph;
}

// Добавление ребра в граф
void addEdge(struct Graph* graph, int src, int dest) {
    struct Node* newNode = createNode(dest);
    newNode->next = graph->adjList[src];
    graph->adjList[src] = newNode;

    newNode = createNode(src);
    newNode->next = graph->adjList[dest];
    graph->adjList[dest] = newNode;
}

// Обход в ширину графа, представленного списками смежности
void bfs(struct Graph* graph, int start) {
    bool visited[G];
    for (int i = 0; i < G; i++) {
        visited[i] = false;
    }
```

```
    int q[G];
    int front = 0, rear = 0;
    visited[start] = true;
    q[rear++] = start;

    while (front != rear) {
        int current = q[front++];
        printf("%d ", current);

        struct Node* temp = graph->adjList[current];
        while (temp != NULL) {
            int adjacentVertex = temp->vertex;
            if (!visited[adjacentVertex]) {
                visited[adjacentVertex] = true;
                q[rear++] = adjacentVertex;
            }
            temp = temp->next;
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    struct Graph* graph = createGraph(G);

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            if (rand() % 2 == 1) {
                addEdge(graph, i, j);
            }
        }
    }

    // Вывод графа в виде списков смежности
    printf("Списки смежности:\n");
    for (int i = 0; i < G; i++) {
        printf("%d: ", i);
        struct Node* temp = graph->adjList[i];
        while (temp != NULL) {
            printf("%d ", temp->vertex);
            temp = temp->next;
        }
        printf("\n");
    }

    // Обход в ширину, начиная с вершины 0
    printf("Обход в ширину: ");
    bfs(graph, 0);
    printf("\n");

    return 0;
}
```

Задание 2*

1. 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.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>

#define G 5  // Размер графа

typedef struct Node {
    int priority; // Приоритет элемента
    int data; // Данные элемента
    struct Node* next; // Указатель на следующий элемент
} Node;

typedef struct PriorityQueue {
    Node* head; // Указатель на голову списка
} PriorityQueue;

// Функция создания новой приоритетной очереди
PriorityQueue* create_queue() {
    PriorityQueue* pq = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    pq->head = NULL; // инициализация головы как пустой
    return pq;
}

// Функция добавления элемента в приоритетную очередь
void enqueue(PriorityQueue* pq, int data, int priority) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->priority = priority; // установка приоритета нового узла
    new_node->data = data;
    new_node->next = NULL;

    // Если очередь пуста либо приоритет нового элемента выше приоритета головы
    if (pq->head == NULL || pq->head->priority < priority) {
        new_node->next = pq->head; // новый узел указ на голову
        pq->head = new_node; // новый узел станов головой
    }
    else {
        // Ищем место для вставки нового элемента
        Node* current = pq->head;
        while (current->next != NULL && current->next->priority >= priority) {
            current = current->next; //переход к следующему
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}
```

```
// Функция извлечения элемента с наивысшим приоритетом
int dequeue(PriorityQueue* pq) {
    if (pq->head == NULL) {
        return -1; // Очередь пуста
    }
    Node* temp = pq->head;
    int data = temp->data;
    pq->head = pq->head->next;
    free(temp); // Освобождаем память для узла
    return data; // Возвращаем данные
}

// Функция для просмотра очереди
void review_queue(PriorityQueue* pq) {
    Node* current = pq->head;
    if (current == NULL) {
        printf("Очередь пуста\n");
        return;
    }

    printf("Содержимое очереди:\n");
    while (current != NULL) {
        printf("Данные - %d, Приоритет - %d\n", current->data, current->priority);
        current = current->next;
    }
}

// Функция проверки, пуста ли очередь
bool is_empty(PriorityQueue* pq) {
    return pq->head == NULL;
}

// Функция освобождения памяти очереди
void free_queue(PriorityQueue* pq) {
    Node* current = pq->head;
    Node* next_node;
    while (current != NULL) {
        next_node = current->next;
        free(current);
        current = next_node;
    }
    free(pq);
}

// Обход в ширину графа, представленного матрицей смежности, с использованием очереди,
построенной на основе структуры данных «список»
void bfs(int M[G][G], int start) {
    bool visited[G]; // массив для отслеживания посещенных узлов
    for (int i = 0; i < G; i++) {
        visited[i] = false; // инициализация всех как непосещенных
    }

    PriorityQueue* pq = create_queue(); // создание приоритетной очереди
    visited[start] = true; // стартовый узел посещен
    enqueue(pq, start, 0); // передача стартового в очередь

    while (!is_empty(pq)) {
        int current = dequeue(pq); // извлекаем узел с наивысшим приоритетом
```

```
        printf("%d ", current);

        for (int i = 0; i < G; i++) { // проход по всем возможным соседям
            if (M[current][i] == 1 && !visited[i]) { // если есть связь и узел непосещен
                visited[i] = true;
                enqueue(pq, i, 0);
            }
        }
    }

    free_queue(pq);
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    int M[G][G]; // Матрица смежности

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0; // Нет петли
            }
            else {
                M[i][j] = 0; // Изначально нет ребер
            }
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Обход в ширину, начиная с вершины 0
    printf("Обход в ширину: ");
    bfs(M, 0);
    printf("\n");

    return 0;
}
```

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

Ответ:

Использует обход в ширину:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>
#include <time.h>

#define G 500  // Размер графа
int M[G][G];      // Матрица смежности
bool visited[G];  // Массив для отслеживания посещенных вершин

void bfs(int start) {
    int q[G];
    int front = 0, rear = 0;
    visited[start] = true;
    q[rear++] = start;

    while (front != rear) {
        int current = q[front++];
        printf("%d ", current);

        for (int i = 0; i < G; i++) {
            if (M[current][i] == 1 && !visited[i]) {
                visited[i] = true;
                q[rear++] = i;
            }
        }
    }
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            if (i == j) {
                M[i][j] = 0;  // Нет петли
            }
            else {
                M[i][j] = 0;  // Изначально нет рёбер
            }
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
```

```
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2;  // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Измерение времени выполнения обхода в ширину
    clock_t start_time = clock();  // Запоминаем время начала
    // Обход в ширину, начиная с вершины 0
    printf("Обход в ширину: ");
    bfs(0);
    printf("\n");
    clock_t end_time = clock();  // Запоминаем время окончания

    // Вычисляем и выводим время выполнения
    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Время выполнения: %f секунд\n", time_taken);

    return 0;
}
```

```
Консоль отладки Microsoft Visual Studio
1 1 0 0 0 0 0 1 1 0 0 0 1 0 0 1 1 0 0 1 0 0 1 1 1 0 1 0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 0 0 1 0 1 1 0 0 0 0 1 1 1 0 1 1 0 0
1 0 1 0 1 0 1 0 1 0 0 0 1 1 0 1 0 0 1 1 0 1 0 1 1 0 1 1 0 1 0 1 0 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1 0 1 0 1 0 1
1 1 0 1 1 0 1 0 1 1 1 0 1 0 0 0 1 1 0 0 1 0 0 1 0 1 1 1 0 1 0 1 1 1 1 0 0 1 1 1 0 1 1 0 0 1 0 0 1 0 0 0 1 1 1 1 0 1 0 1
1 0 1 0 0 0 0 0 0 0 0 1 0 1 1 1 1 1 1 1 0 0 1 0 1 1 0 1 0 1 1 0 0 0 1 1 1 1 1 0 0 0 0 0 0 1 0 1 0 0 1 1 0 1 1 0 0 0 0 0 0
0 0 0 1 0 0 0 1 0 1 1 1 0 0 0 0 0 1 0 0 1 0 1 1 0 1 1 0 0 1 1 0 1 0 0 0 1 1 0 1 1 1 1 0 0 0 0 0 1 0 1 1 0 1 1 1 1 0 0 1
0 1 1 1 0 0 0 1 0 1 1 1 0 1 1 0 1 0 1 0 1 1 1 1 1 1 1 0 1 1 0 0 1 0 1 1 1 1 0 1 1 1 0 0 0 0 0 0 1 0 1 0 1 0 0 0 1 1 1 1 1
1 0 1 1 0 0 0 0 0 1 0 1 0 1 1 0 1 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 1 1 1 0 1 1 0 1 0 0 1 1 1 1 0 0 1 0 1 0 1 0 0
1 1 0 1 1 1 1 1 1 1 1 1 0 0 1 1 0 1 1 0 1 0 0 1 0 1 1 1 0 0 1 0 0 0 1 1 0 1 0 1 1 0 1 0 0 0 1 0 0 1 0 0 0 0 1 1 0 1 0 1
1 0 1 1 0 1 1 1 1 0 0 0 1 0 0 1 0 1 1 0
Обход в ширину: 0 2 3 10 11 13 14 16 17 19 22 25 26 28 29 30 31 33 35 36 37 39 40 41 42 49 51 55 56 57 60 61 65 67 68 73
 76 77 81 84 88 90 92 93 96 99 100 101 103 104 105 107 111 113 114 115 116 123 124 125 126 127 132 134 136 138 141 142 1
43 146 147 149 151 153 154 155 159 161 163 165 169 170 174 176 177 178 179 180 183 185 187 188 189 190 191 192 193 196 1
97 203 204 206 207 208 209 210 211 213 215 218 219 220 221 222 223 227 228 231 232 233 235 238 240 243 245 246 247 248 2
51 252 256 258 260 261 265 268 269 270 273 275 276 278 279 281 285 286 288 290 291 292 293 296 297 299 300 301 302 303 3
05 307 308 309 310 311 314 316 319 321 325 326 328 329 330 332 334 335 339 340 341 346 347 348 349 352 355 357 358 359 3
63 365 366 367 369 371 375 376 380 382 383 384 385 386 388 389 390 392 393 395 396 397 398 399 400 405 411 412 416 418 4
19 421 423 424 427 428 429 432 435 436 437 438 439 442 445 446 453 454 456 459 460 461 464 472 473 474 476 477 478 479 4
85 486 491 493 495 499 1 4 6 7 8 9 12 18 21 23 24 34 43 44 48 52 63 64 66 69 71 75 79 80 83 85 86 87 89 91 97 106 108 10
9 117 120 121 131 133 144 150 156 158 160 162 166 167 173 182 186 195 198 199 212 214 224 225 229 239 241 242 244 263 26
4 267 271 274 282 287 298 304 306 312 315 317 318 322 324 331 333 338 344 350 351 360 361 368 372 373 374 377 379 381 38
7 394 401 402 403 404 406 407 408 409 410 413 414 417 425 426 430 431 433 440 441 443 449 452 462 468 475 480 482 488 49
0 492 494 498 20 27 32 46 47 54 58 59 62 70 72 74 82 95 98 102 110 112 119 122 129 130 137 148 157 175 181 184 200 201 2
17 226 230 234 249 254 255 272 277 280 284 289 313 320 327 337 342 362 370 391 422 444 450 455 458 465 467 469 481 483 4
87 489 496 497 5 15 45 53 140 152 168 171 172 194 216 257 262 266 294 295 323 353 356 378 415 447 448 457 463 466 470 38
 50 94 128 135 164 205 237 283 343 345 354 434 451 471 118 139 250 259 364 484 78 145 236 253 336 202 420
Время выполнения: 0,068000 секунд

C:\Users\dimav\source\repos\ConsoleApplication1\Debug\ConsoleApplication1.exe (процесс 19428) завершил работу с кодом 0.
Нажмите любую клавишу, чтобы закрыть это окно…
```

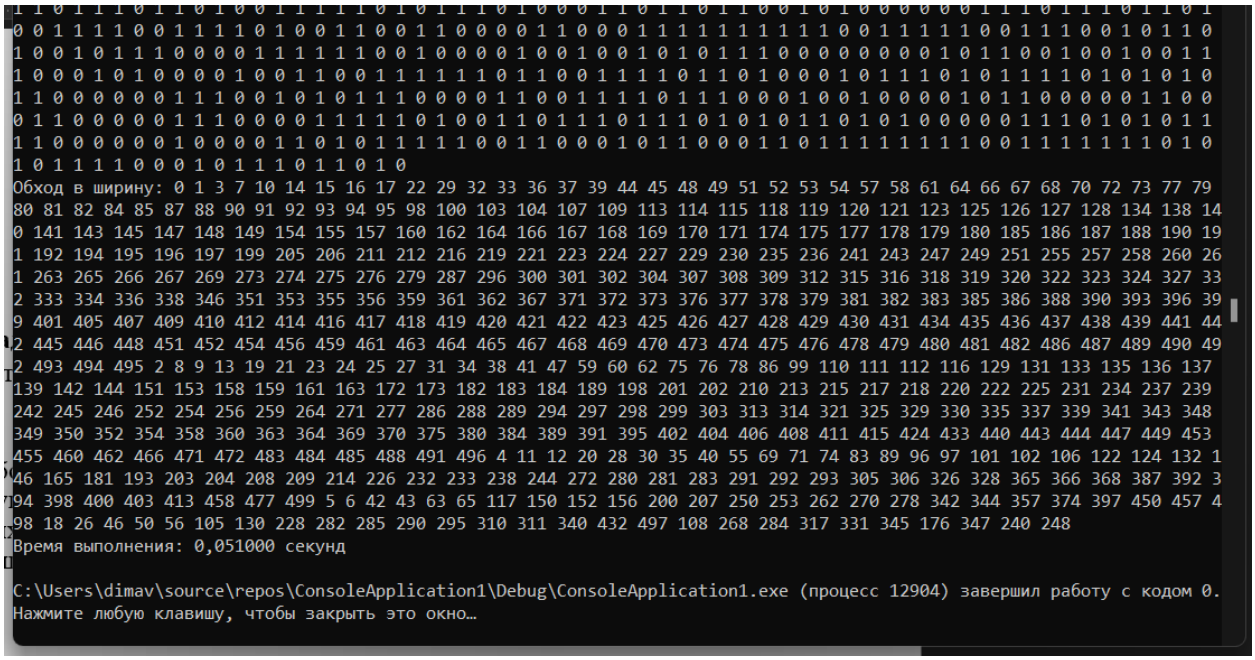
Использует очередь:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
```

```
#include <locale.h>

#define G 500 // Размер графа

typedef struct Node {
    int priority; // Приоритет элемента
    int data; // Данные элемента
    struct Node* next; // Указатель на следующий элемент
} Node;

typedef struct PriorityQueue {
    Node* head; // Указатель на голову списка
} PriorityQueue;

// Функция создания новой приоритетной очереди
PriorityQueue* create_queue() {
    PriorityQueue* pq = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    pq->head = NULL;
    return pq;
}

// Функция добавления элемента в приоритетную очередь
void enqueue(PriorityQueue* pq, int data, int priority) {
    Node* new_node = (Node*)malloc(sizeof(Node));
    new_node->priority = priority;
    new_node->data = data;
    new_node->next = NULL;

    // Если очередь пуста либо приоритет нового элемента выше приоритета головы
    if (pq->head == NULL || pq->head->priority < priority) {
        new_node->next = pq->head;
        pq->head = new_node;
    }
    else {
        // Ищем место для вставки нового элемента
        Node* current = pq->head;
        while (current->next != NULL && current->next->priority >= priority) {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}

// Функция извлечения элемента с наивысшим приоритетом
int dequeue(PriorityQueue* pq) {
    if (pq->head == NULL) {
        return -1; // Очередь пуста
    }
    Node* temp = pq->head;
    int data = temp->data;
    pq->head = pq->head->next;
    free(temp); // Освобождаем память для узла
    return data; // Возвращаем данные
}

// Функция проверки, пуста ли очередь
bool is_empty(PriorityQueue* pq) {
```

```
    return pq->head == NULL;
}

// Функция освобождения памяти очереди
void free_queue(PriorityQueue* pq) {
    Node* current = pq->head;
    Node* next_node;
    while (current != NULL) {
        next_node = current->next;
        free(current);
        current = next_node;
    }
    free(pq);
}

// Обход в ширину графа, представленного матрицей смежности
void bfs(int M[G][G], int start) {
    bool visited[G] = { false }; // Массив для отслеживания посещенных вершин
    PriorityQueue* pq = create_queue();
    visited[start] = true;
    enqueue(pq, start, 0);

    while (!is_empty(pq)) {
        int current = dequeue(pq);
        printf("%d ", current);

        for (int i = 0; i < G; i++) {
            if (M[current][i] == 1 && !visited[i]) {
                visited[i] = true;
                enqueue(pq, i, 0);
            }
        }
    }

    free_queue(pq);
}

int main() {
    setlocale(LC_ALL, "Russian");
    srand(time(NULL));

    int M[G][G]; // Матрица смежности

    // Инициализация матрицы смежности
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            M[i][j] = (i == j) ? 0 : 0; // Нет петель, изначально нет рёбер
        }
    }

    // Генерация случайных рёбер для неориентированного графа
    for (int i = 0; i < G; i++) {
        for (int j = i + 1; j < G; j++) {
            M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
        }
    }

    // Вывод матрицы смежности на экран
```

```
    printf("Матрица смежности:\n");
    for (int i = 0; i < G; i++) {
        for (int j = 0; j < G; j++) {
            printf("%d ", M[i][j]);
        }
        printf("\n");
    }

    // Измерение времени выполнения обхода в ширину
    clock_t start_time = clock();
    printf("Обход в ширину: ");
    bfs(M, 0);
    printf("\n");
    clock_t end_time = clock();

    double time_taken = ((double)(end_time - start_time)) / CLOCKS_PER_SEC;
    printf("Время выполнения: %f секунд\n", time_taken);

    return 0;
}
```



Задание: сделать динамическое выделение памяти и задавать с клавиатуры размер графа, и возможность выбирать стартовую вершину.

Ответ:

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <iostream>
#include <string.h>
#include <locale.h>


void bfs(int start, int G, int** M) {
bool* visited = (bool*)malloc(G * sizeof(bool));
```

```
int* q = (int*)malloc(G * sizeof(int));
int front = 0, rear = 0;
for (int i = 0; i < G; i++) {
visited[i] = false;
}
visited[start] = true;
q[rear++] = start;

while (front != rear) {
int current = q[front++];
printf("%d ", current);

for (int i = 0; i < G; i++) {
if (M[current][i] == 1 && !visited[i]) { // если есть ребро и вершина не посещена
visited[i] = true;
q[rear++] = i;
}
}
}
free(visited);
free(q);
}

int main() {
setlocale(LC_ALL, "Russian");
srand(time(NULL));

int G;
int start;
printf("Введите размер: ");
scanf("%d", &G);
int** M = (int**)malloc(G * sizeof(int*));
for (int i = 0; i < G; i++) {
M[i] = (int*)malloc(G * sizeof(int));
}
// Инициализация матрицы смежности
for (int i = 0; i < G; i++) {
for (int j = 0; j < G; j++) {
if (i == j) {
M[i][j] = 0; // Нет петли
}
else {
M[i][j] = 0; // Изначально нет ребер
}
}
}

// Генерация случайных рёбер для неориентированного графа
for (int i = 0; i < G; i++) {
for (int j = i + 1; j < G; j++) {
M[i][j] = M[j][i] = rand() % 2; // Случайное ребро между i и j
}
}
// Вывод матрицы смежности на экран
printf("Матрица смежности:\n");
for (int i = 0; i < G; i++) {
for (int j = 0; j < G; j++) {
printf("%d ", M[i][j]);
```

```
}
printf("\n");
}

printf("\nВведите стартовую вершину: ");
scanf("%d", &start);

// Обход в ширину, начиная с вершины 0
printf("Обход в ширину: ");
bfs(start, G, M);
printf("\n");
for (int i = 0; i < G; i++) {
free(M[i]);
}
free(M);
return 0;
}
```

Вывод: в ходе выполнения лабораторной работы были выполнены работы с графами, научились генерировать матрицы смежности и осуществлять обход в ширину различными методами. Оценили время работы этих методов и сделали вывод о том, какой из них более эффективен для решения конкретных задач.