南京邮電大學

实 验 报 告

（2024 / 2025 学年 第 一 学期）

课程名称	数据结构
实验名称	图运算
实验时间	2024 年 12 月 11 日
指导单位	计算机学院
指导教师	孙海安

学生姓名	于明宏	班级学号	B23041011
学院(系)	计算机学院	专　　业	信息安全

实 验 报 告

实验名称	**图运算**			**指导教师**	**孙海安**
实验类型	**设计**	**实验学时**	**2**	**实验时间**	**2024.12.11**

一、 实验目的和要求

1. 掌握图的邻接矩阵和邻接表的存储实现方法。
2. 实现图的深度优先和宽度优先遍历运算。
3. 学习使用图算法解决应用问题。

二、实验环境(实验设备)

硬件：微型计算机

软件：Windows 11 Professional Edition 23H2、Microsoft Visual C++ 2022

三、实验原理及内容

题目 1：

1、算法实现

```
#include <stdio.h>           // Include standard input-output header
#include <stdlib.h>          // Include standard library header for dynamic memory functions

// Define status codes for various conditions
#define ERROR 0
#define OK 1
#define Overflow 2
#define Underflow 3
#define NotPresent 4
#define Duplicate 5
```

```
typedef int ElemType;      // Define ElemType as an integer
typedef int Status;         // Define Status as an integer for function return types

// Define structure for adjacency matrix graph
typedef struct {
    ElemType** a;             // Pointer to adjacency matrix
    int n;                       // Number of nodes
    int e;                       // Number of edges
    ElemType noEdge;       // Value representing no edge between nodes
} mGraph;

// Initialize the graph with a given number of nodes and no-edge value
Status Init(mGraph* mg, int nSize, ElemType noEdgeValue) {
    int i, j;
    mg->n = nSize;                                            // Set number of nodes
    mg->e = 0;                                                  // Initialize edge count to zero
    mg->noEdge = noEdgeValue;                        // Set the no-edge indicator
    mg->a = (ElemType**)malloc(nSize * sizeof(ElemType*)); // Allocate memory for adjacency
matrix rows
    if (!mg->a) return ERROR;                        // Return error if memory allocation fails

    for (i = 0; i < mg->n; i++) {
        mg->a[i] = (ElemType*)malloc(nSize * sizeof(ElemType)); // Allocate memory for each
row
        for (j = 0; j < mg->n; j++) {
            mg->a[i][j] = mg->noEdge;         // Initialize all entries to no-edge value
        }
        mg->a[i][i] = 0;                              // Set self-loop weight to zero
    }
    return OK;                                               // Return OK status
}

// Free the memory allocated for the graph
int Destory(mGraph* mg) {
```

```
        int i;
        for (i = 0; i < mg->n; i++) {
            free(mg->a[i]);                    // Free each row in the adjacency matrix
        }
        free(mg->a);                           // Free the matrix pointer itself
        return 1;                              // Return 1 to indicate successful destruction
    }

    // Check if an edge exists between nodes u and v
    Status Exist(mGraph* mg, int u, int v) {
        if (u < 0 || v < 0 || u > mg->n - 1 || v > mg->n - 1 || u == v || mg->a[u][v] == mg->noEdge)
            return ERROR;                      // Return ERROR if edge does not exist
        return OK;                             // Return OK if edge exists
    }

    // Insert an edge with weight w between nodes u and v
    Status Insert(mGraph* mg, int u, int v, ElemType w) {
        if (u < 0 || v < 0 || u > mg->n - 1 || v > mg->n - 1 || u == v)
            return ERROR;                      // Return ERROR if nodes are invalid or are the
same
        if (mg->a[u][v] != mg->noEdge)
            return Duplicate;                  // Return Duplicate if edge already exists
        mg->a[u][v] = w;                       // Set the weight of edge u-v
        mg->e++;                               // Increment edge count
        return OK;                             // Return OK status
    }

    // Remove the edge between nodes u and v
    Status Remove(mGraph* mg, int u, int v) {
        if (u < 0 || v < 0 || u > mg->n - 1 || v > mg->n - 1 || u == v)
            return ERROR;                      // Return ERROR if nodes are invalid or are the
same
        if (mg->a[u][v] == mg->noEdge)
            return NotPresent;                 // Return NotPresent if edge does not exist
        mg->a[u][v] = mg->noEdge;              // Set edge weight to no-edge value to remove it
```

```
    mg->e--;                                        // Decrement edge count
    return OK;                                      // Return OK status
}

// Main function to test graph operations
int main() {
    mGraph g;                                       // Declare a graph variable
    int nSize, edge, u, v, i;                // Declare variables for input
    ElemType w;                                     // Variable to store edge weight

    printf("Please enter the number of nodes in the graph: ");
    scanf_s("%d", &nSize);                     // Input number of nodes
    Init(&g, nSize, -1);                       // Initialize graph with no-edge value as -1

    printf("Please enter the number of edges: ");
    scanf_s("%d", &edge);                      // Input number of edges

    printf("Enter edges with the order of u, v, w: \n");
    for (i = 0; i < edge; i++) {
        scanf_s("%d%d%d", &u, &v, &w);     // Input edge endpoints and weight
        Insert(&g, u, v, w);                 // Insert each edge into the graph
    }

    printf("Please enter the edge to delete:\n");
    printf("Enter the u of the edge: ");
    scanf_s("%d", &u);                         // Input node u for edge deletion

    printf("Enter the v of the edge: ");
    scanf_s("%d", &v);                         // Input node v for edge deletion
    Remove(&g, u, v);                          // Remove the specified edge

    printf("Now searching for the edge just deleted: ");
    if (Exist(&g, u, v))                       // Check if the deleted edge still exists
        printf("OK\n");
    else
```

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

        printf("Now destroying the graph: ");
        if (Destory(&g))                        // Destroy the graph and free memory
            printf("OK\n");
        else
            printf("ERROR\n");

        return 0;                               // End of program
    }
```

2、复杂度分析

（1）Init 函数

时间复杂度：O(n)

该函数用于初始化图，创建一个长度为 n 的指针数组并初始化为 NULL，因此时间复杂度为 O(n)，其中 n 为图的顶点数。

（2）Destory 函数

时间复杂度：O(n + e)

该函数用于释放图的内存。它需要遍历每个顶点对应的邻接链表，并释放链表中的所有边结点。总的复杂度为 O(n + e)，其中 n 是顶点数，e 是边数。

（3）Exist 函数

时间复杂度：O(d)

该函数用于检查图中是否存在指定的边。由于邻接表的查找操作需要遍历邻接链表，时间复杂度为 O(d)，其中 d 是顶点 u 的度数。

（4）Insert 函数

时间复杂度：O(d)

该函数先调用 Exist 函数检查边是否存在，随后在邻接链表的头部插入新边。由于查找操作的复杂度为 O(d)，整体时间复杂度也为 O(d)，其中 d 是顶点 u 的度数。

（5）Remove 函数

时间复杂度：O(d)

该函数用于删除指定的边，需要遍历邻接链表查找边的位置，最坏情况下的时间复杂度为 O(d)，其中 d 是顶点 u 的度数。

（6）main 函数

时间复杂度：O(n + e)

main 函数中调用了 Init、Insert、Remove、Exist 和 Destory 函数，整体时间复杂度取决于这些操作。初始化和销毁的复杂度为 O(n + e)，而插入边操作遍历每个输入的边，因此复杂度也是 O(n

+ e)。

3、实验结果与结论

Please enter the number of nodes in the graph: 3

Please enter the number of edges: 3

Enter edges with the order of u, v, w:

1 2 10

2 3 15

1 3 20

Please enter the edge to delete:

Enter the u of the edge: 1

Enter the v of the edge: 2

Now searching for the edge just deleted: ERROR

Now destroying the graph: OK

题目 2：

1、算法实现

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <stdbool.h>

#define ERROR 0                  // Define error status
#define OK 1                     // Define success status
#define Overflow 2           // Define overflow status
#define Underflow 3          // Define underflow status
#define NotPresent 4       // Define "not present" status
#define Duplicate 5          // Define duplicate status

typedef int ElemType;      // Define element type as integer
typedef int Status;          // Define status type as integer

// Define structure for the graph with adjacency matrix representation
typedef struct {
        ElemType** a;              // Adjacency matrix
        int n;                          // Number of vertices
        int e;                          // Number of edges
```

```
    ElemType noEdge;        // Value representing no edge between vertices
} mGraph;

// Define structure for a circular queue
typedef struct {
    int front;                   // Index of the front element
    int rear;                    // Index of the rear element
    int maxSize;                 // Maximum size of the queue
    ElemType* element;    // Array to store queue elements
} Queue;

// Initialize the queue with a given size
void Create(Queue* Q, int mSize) {
    Q->maxSize = mSize;                                              // Set maximum size of the
queue
    Q->element = (ElemType*)malloc(sizeof(ElemType) * mSize); // Allocate memory for elements
    Q->front = Q->rear = 0;                                          // Initialize front and rear indices
}

// Check if the queue is empty
int IsEmpty(Queue* Q) {
    return Q->front == Q->rear;
}

// Check if the queue is full
int IsFULL(Queue* Q) {
    return (Q->rear + 1) % Q->maxSize == Q->front;
}

// Retrieve the front element of the queue without removing it
int Front(Queue* Q, ElemType* x) {
    if (IsEmpty(Q))
        return 0;
    *x = Q->element[(Q->front + 1) % Q->maxSize];
    return 1;
```

```
}

// Add an element to the rear of the queue
int EnQueue(Queue* Q, ElemType x) {
    if (IsFULL(Q))
        return 0;
    Q->rear = (Q->rear + 1) % Q->maxSize; // Update rear index circularly
    Q->element[Q->rear] = x;                 // Insert element
    return 1;
}

// Remove the front element from the queue
int DeQueue(Queue* Q) {
    if (IsEmpty(Q))
        return 0;
    Q->front = (Q->front + 1) % Q->maxSize; // Update front index circularly
    return 1;
}

// Initialize the graph with a specified size and no-edge value
Status Init(mGraph* mg, int nSize, ElemType noEdgeValue) {
    int i, j;
    mg->n = nSize;                              // Set number of vertices
    mg->e = 0;                                  // Initialize number of edges to 0
    mg->noEdge = noEdgeValue;                   // Set value representing no edge
    mg->a = (ElemType**)malloc(nSize * sizeof(ElemType*)); // Allocate memory for adjacency
matrix
    if (!mg->a) return ERROR;                   // Return error if allocation fails
    for (i = 0; i < mg->n; i++) {
        mg->a[i] = (ElemType*)malloc(nSize * sizeof(ElemType)); // Allocate row
        for (j = 0; j < mg->n; j++) {
            mg->a[i][j] = mg->noEdge;           // Initialize with no-edge value
        }
        mg->a[i][i] = 0;                        // Set self-loop edge weight to 0
    }
```

```
        return OK;
    }

    // Destroy the graph by freeing allocated memory
    int Destroy(mGraph* mg) {
        int i;
        for (i = 0; i < mg->n; i++) {
            free(mg->a[i]);                         // Free each row
        }
        free(mg->a);                                // Free the adjacency matrix
        return 1;
    }

    // Check if an edge exists between two vertices
    Status Exist(mGraph* mg, int u, int v) {
        if (u < 0 || v < 0 || u >= mg->n || v >= mg->n || u == v || mg->a[u][v] == mg->noEdge)
            return ERROR;                           // Return error if edge does not exist
        return OK;
    }

    // Insert an edge between two vertices with a given weight
    Status Insert(mGraph* mg, int u, int v, ElemType w) {
        if (u < 0 || v < 0 || u >= mg->n || v >= mg->n || u == v) return ERROR;
        if (mg->a[u][v] != mg->noEdge) return Duplicate;
        mg->a[u][v] = w;                            // Set weight of edge
        mg->e++;                                    // Increment edge count
        return OK;
    }

    // Remove an edge between two vertices
    Status Remove(mGraph* mg, int u, int v) {
        if (u < 0 || v < 0 || u >= mg->n || v >= mg->n || u == v) return ERROR;
        if (mg->a[u][v] == mg->noEdge) return NotPresent;
        mg->a[u][v] = mg->noEdge;                   // Set to no-edge value
        mg->e--;                                    // Decrement edge count
```

```
    return OK;
}

// Depth-First Search (DFS) starting from a vertex
void DFS(int v, int visited[], mGraph g) {
    int j;
    printf("%d ", v);                              // Print the visited vertex
    visited[v] = 1;                                // Mark vertex as visited
    for (j = 0; j < g.n; j++) {
        if (!visited[j] && g.a[v][j] > 0) {
            DFS(j, visited, g);                    // Recursively visit adjacent vertices
        }
    }
}

// Perform DFS on the entire graph
void DFSGraph(mGraph g) {
    int i;
    int* visited = (int*)malloc(g.n * sizeof(int)); // Allocate memory for visited array
    for (i = 0; i < g.n; i++) {
        visited[i] = 0;                            // Initialize all vertices as unvisited
    }
    for (i = 0; i < g.n; i++) {
        if (!visited[i]) {
            DFS(i, visited, g);                    // Start DFS if vertex is unvisited
        }
    }
    free(visited);                                 // Free visited array
}

// Breadth-First Search (BFS) starting from a vertex
void BFS(int v, int visited[], mGraph g) {
    Queue q;
    Create(&q, g.n);                               // Initialize queue
    visited[v] = 1;                                // Mark vertex as visited
```

```
    printf("%d ", v);                                   // Print the visited vertex
    EnQueue(&q, v);                                     // Enqueue the starting vertex
    while (!IsEmpty(&q)) {
        Front(&q, &v);                                  // Get front vertex
        DeQueue(&q);                                    // Remove front vertex from queue
        for (int i = 0; i < g.n; i++) {
            if (!visited[i] && g.a[v][i] > 0) {
                visited[i] = 1;                         // Mark as visited
                printf("%d ", i);                       // Print the visited vertex
                EnQueue(&q, i);                         // Enqueue adjacent vertex
            }
        }
    }
}

// Perform BFS on the entire graph
void BFSGraph(mGraph g) {
    int i;
    int* visited = (int*)malloc(g.n * sizeof(int)); // Allocate memory for visited array
    for (i = 0; i < g.n; i++) {
        visited[i] = 0;                                 // Initialize all vertices as unvisited
    }
    for (i = 0; i < g.n; i++) {
        if (!visited[i]) {
            BFS(i, visited, g);                         // Start BFS if vertex is unvisited
        }
    }
    free(visited);                                      // Free visited array
}

int main() {
    mGraph g;
    int nSize, edge, u, v, i;
    ElemType w;
    printf("Please enter the size of the graph: ");
```

```c
    scanf_s("%d", &nSize);                          // Input graph size
    Init(&g, nSize, -1);                            // Initialize graph
    printf("Please enter the number of the edges: ");
    scanf_s("%d", &edge);                           // Input number of edges

    printf("Enter edges with the order of u, v, w: \n");
    for (i = 0; i < edge; i++) {
        scanf_s("%d%d%d", &u, &v, &w);              // Input each edge's vertices and weight
        Insert(&g, u, v, w);                        // Insert edge into graph
    }
    printf("DFS:\n");
    DFSGraph(g);                                    // Perform DFS
    printf("\nBFS:\n");
    BFSGraph(g);                                    // Perform BFS

    Destroy(&g);                                    // Free graph memory
    return 0;
}
```

2、复杂度分析

（1）Create 函数

时间复杂度：O(1)

该函数用于初始化队列，分配固定大小的数组，初始化前后指针，时间复杂度为 O(1)。

（2）IsEmpty 函数

时间复杂度：O(1)

该函数用于检查队列是否为空，只需判断前后指针是否相等，因此时间复杂度为 O(1)。

（3）IsFULL 函数

时间复杂度：O(1)

该函数用于检查队列是否已满，计算方式简单，仅需一次取模运算，因此时间复杂度为 O(1)。

（4）Front 函数

时间复杂度：O(1)

该函数用于获取队列头部元素，无需遍历元素，时间复杂度为 O(1)。

（5）EnQueue 函数

时间复杂度：O(1)

该函数用于向队列末尾添加元素，操作简单，仅涉及指针操作，时间复杂度为 O(1)。

（6）DeQueue 函数

时间复杂度：O(1)

该函数用于删除队列头部元素，仅涉及指针操作，时间复杂度为 O(1)。

（7）Init 函数

时间复杂度：O(n^2)

该函数初始化图的邻接矩阵，初始化每个元素的值（包含无边值和自环），因此复杂度为 O(n^2)，其中 n 是顶点数。

（8）Destroy 函数

时间复杂度：O(n)

该函数用于释放图的内存，只需遍历顶点行并释放相应的内存，因此时间复杂度为 O(n)。

（9）Exist 函数

时间复杂度：O(1)

该函数用于检查两个顶点之间是否存在边，访问邻接矩阵的指定位置，复杂度为 O(1)。

（10）Insert 函数

时间复杂度：O(1)

该函数用于插入边，直接设置邻接矩阵的相应元素，时间复杂度为 O(1)。

（11）Remove 函数

时间复杂度：O(1)

该函数用于删除边，只需访问并更新邻接矩阵的相应元素，因此时间复杂度为 O(1)。

（12）DFS 函数

时间复杂度：O(n + e)

该函数使用递归方式进行深度优先搜索，每个顶点和每条边访问一次，总的时间复杂度为 O(n + e)，其中 n 是顶点数，e 是边数。

（13）DFSGraph 函数

时间复杂度：O(n + e)

该函数用于遍历图中所有顶点并进行 DFS，每个顶点和每条边访问一次，时间复杂度为 O(n + e)。

（14）BFS 函数

时间复杂度：O(n + e)

该函数使用队列进行广度优先搜索，每个顶点和每条边访问一次，总的时间复杂度为 O(n + e)。

（15）BFSGraph 函数

时间复杂度：O(n + e)

该函数用于遍历图中所有顶点并进行 BFS，每个顶点和每条边访问一次，总的时间复杂度为 O(n + e)。

（16）main 函数

时间复杂度：O(n^2)

main 函数中包含图的初始化、边的插入以及 DFS 和 BFS 遍历操作，其中 DFS 和 BFS 的复杂度为 O(n + e)，插入边的循环复杂度为 O(n^2)。

3、实验结果与结论

Please enter the size of the graph: 6

Please enter the number of the edges: 10

Enter edges with the order of u, v, w:

5 1 1

5 3 1

1 2 1

1 3 1

3 2 1

2 0 1

0 1 1

4 0 1

4 2 1

3 0 1

DFS:

0 1 2 3 4 5

BFS:

0 1 2 3 4 5

题目 3：

1、算法实现

```
#include<stdio.h>
#include<stdlib.h>
#include <windows.h>
#define ERROR 0
#define OK 1
#define Overflow 2          // Indicates overflow
#define Underflow 3         // Indicates underflow
#define NotPresent 4      // Indicates element does not exist
#define Duplicate 5       // Indicates duplicate element
typedef int ElemType;
typedef int Status;
```

```
// Definition of the adjacency list structure
typedef struct ENode {
    int adjVex;                 // Adjacent vertex of any vertex u
    ElemType w;                 // Weight of the edge
    struct ENode* nextArc;   // Pointer to the next edge node
} ENode;

typedef struct {
    int n;                      // Current number of vertices in the graph
    int e;                      // Current number of edges in the graph
    ENode** a;                  // Pointer to a 1D array of pointers
} LGraph;


// Initialization of the adjacency list
Status Init(LGraph* lg, int nSize) {
    int i;
    lg->n = nSize;   // Set the number of vertices
    lg->e = 0;         // Initialize the number of edges to 0
    lg->a = (ENode**)malloc(nSize * sizeof(ENode*));   // Dynamically create a 1D array of
pointers with length n
    if (!lg->a) return ERROR; // Check if memory allocation is successful
    else {
        for (i = 0; i < lg->n; i++) {
            lg->a[i] = NULL;   // Initialize each element in the pointer array to NULL
        }
        return OK;
    }
}


// Destroy the adjacency list (changed to int type to return a value)
int Destory(LGraph* lg) {
    int i;
    ENode* p, * q;
```

```
        for (i = 0; i < lg->n; i++) { // Loop through each vertex
            p = lg->a[i];            // Pointer p points to the first edge node of the vertex i’s linked list
            q = p;
            while (p) {              // Free all edge nodes in the linked list of vertex i
                p = p->nextArc;
                free(q);
                q = p;
            }
        }
        free(lg->a);                 // Free the memory of the 1D pointer array a
        return 1;                    // Return a value, as it's an int function
    }


    // Search for an edge in the adjacency list
    Status Exist(LGraph* lg, int u, int v) {
        ENode* p;
        if (u < 0 || v < 0 || u > lg->n - 1 || v > lg->n - 1 || u == v) return ERROR; // Validate vertices u and v
        p = lg->a[u];                // Pointer p points to the first edge node of the vertex u’s linked list
        while (p != NULL && p->adjVex != v) { // Traverse to find if there is an edge between u and v
            p = p->nextArc;
        }
        if (!p) return ERROR;        // If edge is not found, return ERROR
        else return OK;
    }


    // Insert an edge into the adjacency list
    Status Insert(LGraph* lg, int u, int v, ElemType w) {
        ENode* p;
        if (u < 0 || v < 0 || u > lg->n - 1 || v > lg->n - 1 || u == v) return ERROR; // Validate vertices u and v
```

```
        if (Exist(lg, u, v)) return Duplicate;    // If edge already exists, return Duplicate error
        p = (ENode*)malloc(sizeof(ENode)); // Allocate memory for a new edge node
        p->adjVex = v;                                  // Set the adjacent vertex to v
        p->w = w;                                        // Set the weight of the edge
        p->nextArc = lg->a[u];                       // Insert the new edge node at the beginning of the linked
list
        lg->a[u] = p;
        lg->e++;                                         // Increase the edge count
        return OK;
    }



    // Remove an edge from the adjacency list
    Status Remove(LGraph* lg, int u, int v) {
        ENode* p, * q;
        if (u < 0 || v < 0 || u > lg->n - 1 || v > lg->n - 1 || u == v) return ERROR; // Validate vertices u and
v
        p = lg->a[u];
        q = NULL;
        while (p && p->adjVex != v) {       // Search to check if the edge to be deleted exists
            q = p;
            p = p->nextArc;
        }
        if (!p) return NotPresent;            // If p is NULL, the edge does not exist
        if (q) q->nextArc = p->nextArc;    // Remove the edge from the linked list
        else lg->a[u] = p->nextArc;
        free(p);                                        // Free memory of the edge node
        lg->e--;                                        // Decrease the edge count
        return OK;
    }



    int main() {
        LGraph g;                                         // Declare graph variable
        int nSize, edge, u, v, i;                     // Declare variables for input
```

```
ElemType w;                                    // Variable to store edge weight

printf("Please enter the number of nodes in the graph: ");
scanf_s("%d", &nSize);                         // Input number of nodes
Init(&g, nSize);                               // Initialize graph

printf("Please enter the number of edges: ");
scanf_s("%d", &edge);                          // Input number of edges

printf("Enter edges with the order of u, v, w: \n");
for (i = 0; i < edge; i++) {
        scanf_s("%d%d%d", &u, &v, &w);         // Input edge endpoints and weight
        Insert(&g, u, v, w);                   // Insert each edge into the graph
}

printf("Please enter the edge to delete:\n");
printf("Enter the u of the edge: ");
scanf_s("%d", &u);                             // Input node u for edge deletion

printf("Enter the v of the edge: ");
scanf_s("%d", &v);                             // Input node v for edge deletion
Remove(&g, u, v);                              // Remove the specified edge

printf("Now searching for the edge just deleted: ");
if (Exist(&g, u, v))                           // Check if the deleted edge still exists
        printf("OK\n");
else
        printf("ERROR\n");

printf("Now destroying the graph: ");
if (Destory(&g))                               // Destroy the graph and free memory
        printf("OK\n");
else
        printf("ERROR\n");
```

```
    return 0;                          // End of program
}
```

2、复杂度分析

（1）Init 函数

时间复杂度：O(n)

该函数用于初始化邻接表图的顶点数量，并将每个指针初始化为 NULL，因此时间复杂度为 O(n)，其中 n 为顶点数。

（2）Destory 函数

时间复杂度：O(n + e)

该函数遍历每个顶点的邻接链表并释放所有边结点的内存，因此时间复杂度为 O(n + e)，其中 n 是顶点数，e 是边数。

（3）Exist 函数

时间复杂度：O(deg(u))

该函数用于检查从顶点 u 到 v 是否存在边，需遍历 u 的邻接链表，因此时间复杂度为 O(deg(u))，其中 deg(u) 是顶点 u 的度数。

（4）Insert 函数

时间复杂度：O(deg(u))

在插入新边前需检查该边是否已存在，调用 Exist 函数，因此总时间复杂度为 O(deg(u))。

（5）Remove 函数

时间复杂度：O(deg(u))

该函数遍历顶点 u 的邻接链表查找目标边，因此时间复杂度为 O(deg(u))。

（6）main 函数

时间复杂度：O(n + e)

在 main 函数中，调用了图的初始化、边的插入、边的删除和边的存在性检查，总的时间复杂度为 O(n + e)，因为所有操作的复杂度都受顶点数和边数的影。。

3、实验结果与结论

```
Please enter the number of nodes in the graph: 3
Please enter the number of edges: 3
Enter edges with the order of u, v, w:
1 2 10
2 3 15
1 3 20
Please enter the edge to delete:
Enter the u of the edge: 1
Enter the v of the edge: 2
```

Now searching for the edge just deleted: ERROR

Now destroying the graph: OK

题目 4：

1、算法实现

```
#include<stdio.h>
#include<stdlib.h>
#include <windows.h>
#define ERROR 0
#define OK 1
#define Overflow 2          // Overflow indicator
#define Underflow 3         // Underflow indicator
#define NotPresent 4        // Element not present indicator
#define Duplicate 5         // Duplicate element indicator
typedef int ElemType;
typedef int Status;

// Structure definition for adjacency list node
typedef struct ENode {
    int adjVex;                  // Adjacent vertex of any vertex u
    ElemType w;                  // Weight of the edge
    struct ENode* nextArc;       // Pointer to the next edge node
} ENode;

// Structure definition for graph using adjacency list
typedef struct {
    int n;              // Current number of vertices in the graph
    int e;              // Current number of edges in the graph
    ENode** a;          // Pointer to an array of pointers (adjacency list)
} LGraph;

// Structure definition for circular queue
typedef struct {
    int front;          // Front of the queue
    int rear;           // Rear of the queue
    int maxSize;        // Maximum capacity of the queue
```

```
        ElemType* element; // Array to hold queue elements
    } Queue;

    // Creates an empty queue that can hold mSize elements
    void Create(Queue* Q, int mSize) {
        Q->maxSize = mSize;
        Q->element = (ElemType*)malloc(sizeof(ElemType) * mSize);
        Q->front = Q->rear = 0;
    }

    // Checks if the queue is empty, returns TRUE if empty, otherwise FALSE
    BOOL IsEmpty(Queue* Q) {
        return Q->front == Q->rear;
    }

    // Checks if the queue is full, returns TRUE if full, otherwise FALSE
    BOOL IsFULL(Queue* Q) {
        return (Q->rear + 1) % Q->maxSize == Q->front;
    }

    // Gets the front element of the queue and returns it through x. Returns TRUE if successful, otherwise
FALSE
    BOOL Front(Queue* Q, ElemType* x) {
        if (IsEmpty(Q))         // Handle empty queue
            return FALSE;
        *x = Q->element[(Q->front + 1) % Q->maxSize];
        return TRUE;
    }

    // Inserts element x at the rear of queue Q. Returns TRUE if successful, otherwise FALSE
    BOOL EnQueue(Queue* Q, ElemType x) {
        if (IsFULL(Q))          // Handle overflow
            return FALSE;
        Q->rear = (Q->rear + 1) % Q->maxSize;
        Q->element[Q->rear] = x;
```

```
        return TRUE;
    }

    // Deletes the front element of queue Q. Returns TRUE if successful, otherwise FALSE
    BOOL DeQueue(Queue* Q) {
        if (IsEmpty(Q)) {      // Handle empty queue
            return FALSE;
        }
        Q->front = (Q->front + 1) % Q->maxSize;
        return TRUE;
    }

    // Initializes the adjacency list
    Status Init(LGraph* lg, int nSize) {
        int i;
        lg->n = nSize;
        lg->e = 0;
        lg->a = (ENode**)malloc(nSize * sizeof(ENode*)); // Dynamically create an array of pointers
with length n
        if (!lg->a) return ERROR;
        else {
            for (i = 0; i < lg->n; i++) {
                lg->a[i] = NULL;    // Initialize the adjacency list to NULL
            }
            return OK;
        }
    }

    // Searches for an edge in the adjacency list
    Status Exist(LGraph* lg, int u, int v) {
        ENode* p;
        if (u < 0 || v < 0 || u > lg->n - 1 || v > lg->n - 1 || u == v) return ERROR;
        p = lg->a[u];      // Pointer p points to the first edge node of vertex u's adjacency list
        while (p && p->adjVex != v) {
            p = p->nextArc;
```

```
    }
    if (!p) return ERROR; // If edge is not found, return ERROR
    else return OK;
}

// Inserts an edge into the adjacency list
Status Insert(LGraph* lg, int u, int v, ElemType w) {
    ENode* p;
    if (u < 0 || v < 0 || u > lg->n - 1 || v > lg->n - 1 || u == v) return ERROR;
    if (Exist(lg, u, v)) return Duplicate; // If edge exists, return Duplicate error
    p = (ENode*)malloc(sizeof(ENode));      // Allocate memory for the new edge node
    p->adjVex = v;
    p->w = w;
    p->nextArc = lg->a[u];                  // Insert the new edge node at the beginning of the list
    lg->a[u] = p;
    lg->e++;                                // Increment the edge count
    return OK;
}

// DFS traversal for a single vertex in the adjacency list
void DFS(int v, int visited[], LGraph g) {
    ENode* w;
    printf("%d ", v);                       // Visit vertex v
    visited[v] = 1;                         // Mark vertex v as visited
    for (w = g.a[v]; w; w = w->nextArc) {   // Traverse all adjacent vertices of v
        if (!visited[w->adjVex]) {
            DFS(w->adjVex, visited, g);     // Recursively call DFS for unvisited adjacent vertices
        }
    }
}

// DFS traversal for the entire graph in the adjacency list
void DFSGraph(LGraph g) {
    int i;
    int* visited = (int*)malloc(g.n * sizeof(int)); // Dynamically create the visited array
```

```
        for (i = 0; i < g.n; i++) {
            visited[i] = 0; // Initialize visited array
        }
        for (i = 0; i < g.n; i++) {                // Check each vertex, if not visited, call DFS
            if (!visited[i]) {
                DFS(i, visited, g);
            }
        }
        free(visited);                             // Free the visited array
    }

    // BFS traversal for a single vertex in the adjacency list
    void BFS(int v, int visited[], LGraph g) {
        ENode* w;
        Queue q;
        Create(&q, g.n);                           // Initialize queue
        visited[v] = 1;                            // Mark vertex v as visited
        printf("%d ", v);                          // Visit vertex v
        EnQueue(&q, v);                            // Enqueue vertex v
        while (!IsEmpty(&q)) {
            Front(&q, &v);
            DeQueue(&q);                           // Dequeue the front vertex
            for (w = g.a[v]; w; w = w->nextArc) { // Traverse all adjacent vertices of v
                if (!visited[w->adjVex]) {     // If adjacent vertex is not visited, visit and enqueue it
                    visited[w->adjVex] = 1;
                    printf("%d ", w->adjVex);
                    EnQueue(&q, w->adjVex);
                }
            }
        }
    }

    // BFS traversal for the entire graph in the adjacency list
    void BFSGraph(LGraph g) {
        int i;
```

```
    int* visited = (int*)malloc(g.n * sizeof(int)); // Dynamically create the visited array
    for (i = 0; i < g.n; i++) {                    // Initialize visited array
        visited[i] = 0;
    }
    for (i = 0; i < g.n; i++) {                    // Check each vertex, if not visited, call BFS
        if (!visited[i]) {
            BFS(i, visited, g);
        }
    }
    free(visited);
}

int main() {
    LGraph g;
    int i, u, v, enode, edge;
    ElemType w;
    printf("Please enter the number of the nodes: ");
    scanf_s("%d", &enode);            // Read the number of nodes
    Init(&g, enode);                  // Initialize the graph
    printf("Please enter the number of the edges: ");
    scanf_s("%d", &edge);             // Read the number of edges
    printf("Enter edges with the order of u, v, w: \n");
    for (i = 0; i < edge; i++) {      // Read edges and add them to the graph
        scanf_s("%d%d%d", &u, &v, &w);
        Insert(&g, u, v, w);
    }
    printf("DFS:\n");
    DFSGraph(g);                      // Perform DFS traversal
    printf("\nBFS:\n");
    BFSGraph(g);                      // Perform BFS traversal
    return 0;
}
```

2、复杂度分析

（1）Init 函数

时间复杂度：O(n)

该函数用于初始化邻接表图的顶点数量，并将每个指针初始化为 NULL，因此时间复杂度为 O(n)，其中 n 为顶点数。

（2）Exist 函数

时间复杂度：O(deg(u))

该函数用于检查从顶点 u 到 v 是否存在边，需遍历 u 的邻接链表，因此时间复杂度为 O(deg(u))，其中 deg(u) 是顶点 u 的度数。

（3）Insert 函数

时间复杂度：O(deg(u))

在插入新边前需检查该边是否已存在，调用 Exist 函数，因此总时间复杂度为 O(deg(u))。

（4）DFS 函数

时间复杂度：O(n + e)

该函数使用深度优先搜索遍历图的所有顶点和边，每个顶点和每条边仅访问一次，因此时间复杂度为 O(n + e)，其中 n 为顶点数，e 为边数。

（5）DFSGraph 函数

时间复杂度：O(n + e)

该函数调用 DFS 函数对未访问的每个顶点执行深度优先搜索，因此其时间复杂度为 O(n + e)。

（6）BFS 函数

时间复杂度：O(n + e)

该函数使用广度优先搜索遍历图的所有顶点和边，每个顶点和每条边仅访问一次，因此时间复杂度为 O(n + e)。

（7）BFSGraph 函数

时间复杂度：O(n + e)

该函数调用 BFS 函数对未访问的每个顶点执行广度优先搜索，因此其时间复杂度为 O(n + e)。

（8）main 函数

时间复杂度：O(n + e)

在 main 函数中，调用了图的初始化、边的插入、深度优先搜索和广度优先搜索操作，总的时间复杂度为 O(n + e)。

3、实验结果与结论

Please enter the number of the nodes: 4

Please enter the number of the edges: 4

Enter edges with the order of u, v, w:

0 3 1

0 1 1

2 3 1

2 1 1

DFS:

0 1 3 2

BFS:

0 1 3 2

题目 5：

1、算法实现

```
#include <stdio.h>
#include <string.h>

// Variables to store the number of cities and bus routes
int n, m;

// Adjacency matrix to represent connections between cities
int s[50][50];

// Dijkstra's algorithm function to find the minimum number of transfers
int Dijkstra(int start, int end)
{
    int i = 0, j = 0, k = 0; // Loop counters and index variable
    int min; // Minimum distance
    int distance[100]; // Distance array to store distances from the start node
    int visited[100]; // Array to mark visited nodes

    // Initialize the distance and visited arrays with zeros
    memset(distance, 0, sizeof(distance));
    memset(visited, 0, sizeof(visited));

    // Set initial distances from start node; 1 means reachable, 9999 means unreachable
    for (i = 0; i < n; i++)
        distance[i] = s[start][i];

    // Loop to visit each node in the graph
    for (i = 1; i <= n - 1; i++)
    {
```

```
        min = 99999; // Initialize minimum to a large value
        for (j = 0; j < n; j++) {
            // Find the node with the minimum distance that has not been visited
            if (distance[j] < min && !visited[j])
            {
                k = j; // Update k to the current node index
                min = distance[j]; // Update minimum distance
            }
        }
        visited[k] = 1; // Mark the selected node as visited

        // Update distances for adjacent nodes
        for (j = 0; j < n; j++)
            // Find a shorter path if available
            if (distance[j] > distance[k] + s[k][j])
                distance[j] = distance[k] + s[k][j];
    }

    // Number of transfers is the number of visited stations - 1
    return distance[end] - 1;
}

int main()
{
    int i, j, a, b, ans; // Variables for loops and input/output
    memset(s, 0, sizeof(s)); // Initialize the adjacency matrix with zeros

    // Get the number of cities from the user
    printf("Enter the number of cities: ");
    scanf_s("%d", &n);

    // Initialize the adjacency matrix with distances
    for (i = 0; i < n; i++)
        for (j = 0; j < n; j++)
            if (i == j)
```

```
                s[i][j] = 0; // Distance from a city to itself is 0
            else
                s[i][j] = 99999; // Unreachable cities have a large distance

    // Get the number of bus routes from the user
    printf("Enter the number of bus routes: ");
    scanf_s("%d", &m);

    // Get pairs of connected stations
    printf("Enter the two reachable stations:\n");
    for (i = 0; i < m; i++)
    {
        scanf_s("%d%d", &a, &b); // Read the two cities connected by a bus route
        s[a][b] = 1; // Mark the route as reachable (distance = 1)
    }

    // Get the start and end cities from the user
    printf("Enter the start and end points:\n");
    scanf_s("%d%d", &a, &b);

    // Calculate the minimum number of transfers needed
    ans = Dijkstra(a, b);

    // Output the result based on the number of transfers found
    if (ans == 99998)
        printf("Cannot reach destination.\n"); // If unreachable, print a message
    else if (ans > 0)
        printf("Minimum number of transfers: %d\n", ans); // Print the minimum number of
transfers
    else if (ans == 0)
        printf("No transfer needed.\n"); // If no transfers are needed, print a message

    return 0; // End of program
}
```

2、复杂度分析

（1）Dijkstra 函数

时间复杂度：O(n^2)

该函数实现了基于邻接矩阵的 Dijkstra 算法，用于计算从起始点到终点的最小转车次数。函数在每次迭代中都会寻找当前最小距离的未访问顶点，这一步需要 O(n)的时间复杂度。找到顶点后，需要更新该顶点所有相邻顶点的最短距离，这一步的复杂度也是 O(n)。因为需要在所有 n 个顶点上进行这样的操作，所以整体时间复杂度为 O(n^2)。

（2）main 函数

时间复杂度：O(n^2)

在 main 函数中，涉及多个步骤的操作会对复杂度产生影响。初始化邻接矩阵，这部分为双重循环操作，时间复杂度为 O(n^2)。通过循环输入公交线路信息，构建邻接矩阵，时间复杂度为 O(m)，其中 m 是公交线路的数量。调用 Dijkstra 函数进行最小转车次数的计算，复杂度为 O(n^2)。

综上，main 函数的整体时间复杂度为 O(n^2)。

3、实验结果与结论

Enter the number of cities: 4

Enter the number of bus routes: 3

Enter the two reachable stations:

1 2

0 1

2 3

Enter the start and end points:

1 3

Minimum number of transfers: 1

四、实验小结（包括问题和解决方法、心得体会、意见与建议等）

（一）实验中遇到的主要问题及解决方法

1. 问题：在实现图的邻接矩阵和邻接表时，内存分配失败导致程序崩溃。

解决方法：检查内存分配语句，确保在分配失败时有相应的错误处理机制，如返回错误代码或释放已分配的内存。

2. 问题：图的深度优先遍历（DFS）和宽度优先遍历（BFS）结果与预期不符。

解决方法：通过增加调试信息，逐步跟踪遍历过程中的节点访问顺序，发现逻辑错误并修正，确保遍历算法的正确性。

3.问题：在 Dijkstra 算法实现中，对于无连接的图，算法无法正确处理。 解决方法：在算法开始前增加对图的连通性检查，如果图不连通，则提前返回错误信息或特殊值，避免算法执行过程中出现异常。

（二）实验心得

过本次图运算实验，我深刻体会到了图数据结构在算法实现中的重要性。在实现图的邻接矩阵和邻接表存储方法时，我学习到了如何在内存中有效地表示图结构，以及如何通过指针和数组操作来管理图的边信息。此外，通过实现图的深度优先和宽度优先遍历算法，我加深了对这两种基本图遍历方法的理解，掌握了它们在不同场景下的应用。

（三）意见与建议（没有可省略）

可以提供更多的时间上机操作，以确保更多程序设计思路得以实现，提升面向对象语言的掌握程度和编程能力。

五、支撑毕业要求指标点

《数据结构》课程支撑毕业要求的指标点为：

1.2-M 掌握计算机软硬件相关工程基础知识，能将其用于分析计算机及应用领域的相关工程问题。

3.2-H 能够根据用户需求，选取适当的研究方法和技术手段，确定复杂工程问题的解决方案。

4.2-H 能够根据实验方案，配置实验环境、开展实验，使用定性或定量分析方法进行数据分析与处理，综合实验结果以获得合理有效的结论。

实验内容	支撑点 1.2	支撑点 3.2	支撑点 4.2
线性表及多项式的运算	√		
二叉树的基本操作及哈夫曼编码译码系统的实现		√	√
图的基本运算及智能交通中的最佳路径选择问题		√	√
各种内排序算法的实现及性能比较	√		√

六、指导教师评语（含学生能力达成度的评价）

如评分细则所示

成　绩	XX	批阅人	XX	日　期	XXX

	评分项	优秀	良好	中等	合格	不合格
评分细则	遵守实验室规章制度					
	学习态度					
	算法思想准备情况					
	程序设计能力					
	解决问题能力					
	课题功能实现情况					
	算法设计合理性					
	算法效能评价					
	回答问题准确度					
	报告书写认真程度					
	内容详实程度					
	文字表达熟练程度					
	其它评价意见					
	本次实验能力达成评价（总成绩）					