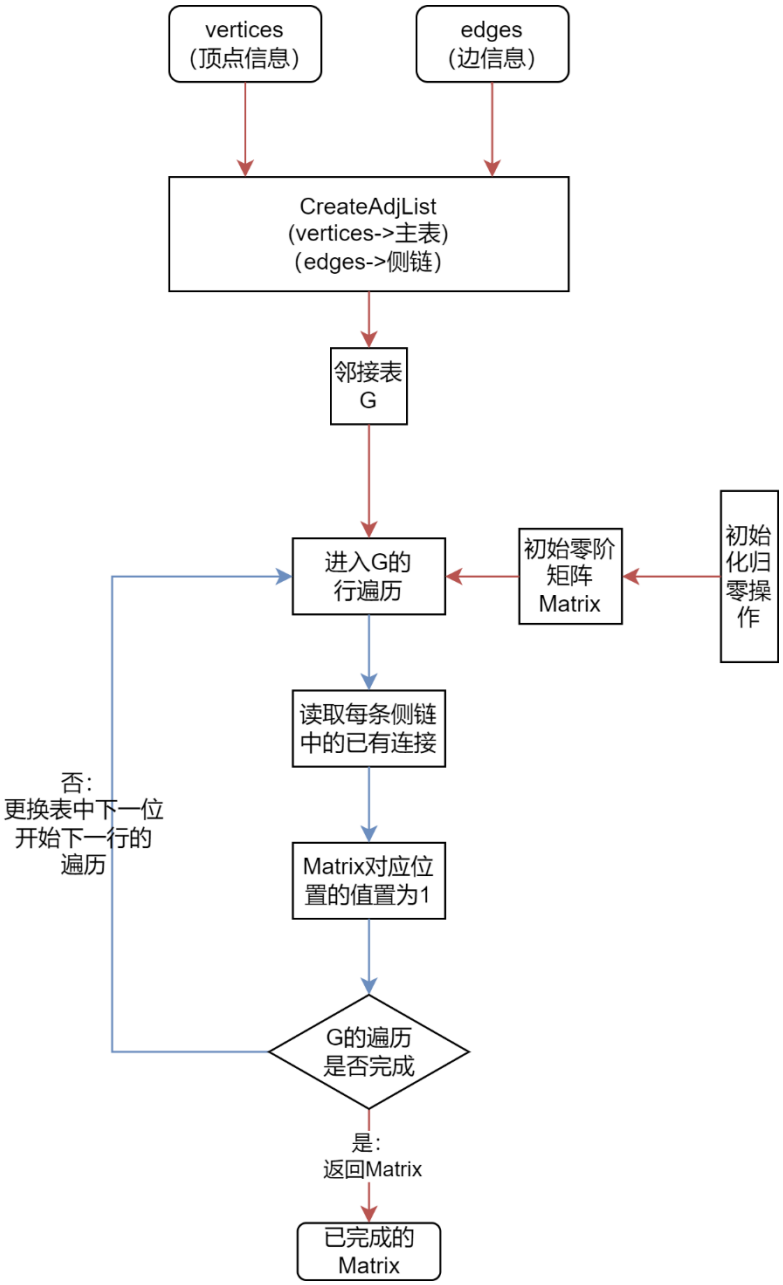
第七章上机报告

吴承洋　22009100500

2. 写出一个无向图的邻接表转换成邻接矩阵的算法。

一、 程序流程图：



二、 程序模块简介：

1. 定义邻接表模块：

（1）子程序功能：

这段代码定义了基础的数据结构来表示图，即为 vexnode 结构体用于表示图的顶点及其邻接表，以及 edgenode 结构体用于表示边表中的节点。

（2）子程序编程思路：

这段代码的编程思路是通过定义结构体来表示图的基本单元以及其间的联系，其中 vexnode 结构体用于表示图的顶点，每个顶点包含了该顶点的信息和一个指向边表的指针。而 edgenode 结构体用于表示边表中的节点，其中包含了邻接顶点的索引以及指向下一个邻接节点的指针。

（3）输入参数：（无）

（4）输出参数：（无）

（5）调用函数：（无）

（6）自测结果：（无）

（7）子程序代码：

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

#define MAX_VERTEX_NUM 100 // 定义图的最大顶点数

typedef char vextype;
typedef struct node
{
 int adjvex;
 struct node* next;
} edgenode;

typedef struct
{
 vextype vertex;
 edgenode* link;
} vexnode;
```

2. 邻接表转邻接矩阵核心模块：

（1）子程序功能：

该子程序的功能是将给定的邻接表表示的图转换为邻接矩阵表示。邻接矩阵是一种图的表示方法，其中矩阵的行和列分别代表图的顶点，矩阵中的值表示对应顶点之间是否存在边。

（2）子程序编程思路：

首先，我们需要初始化邻接矩阵:使用两个嵌套的循环遍历矩阵，将所有元素初始化为 0，表示初始状态下没有边相连。

之后，我们可以将邻接表转换为邻接矩阵:遍历邻接表数组，对于每个顶点，遍历其邻接链表，将对应的邻接顶点在邻接矩阵中的位置标记为 1，表示存在边。

最终，在遍历完成退出循环之后也就完成了对于输出的邻接矩阵的修改。

（3）输入参数：

a. vexnode G[]:邻接表表示的图，其中每个元素是一个顶点，包含该顶点的信息以及指向邻接链表的指针;

b. int n: 图中顶点的数量;

c. int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]: 用于存储邻接矩阵的二维数组。

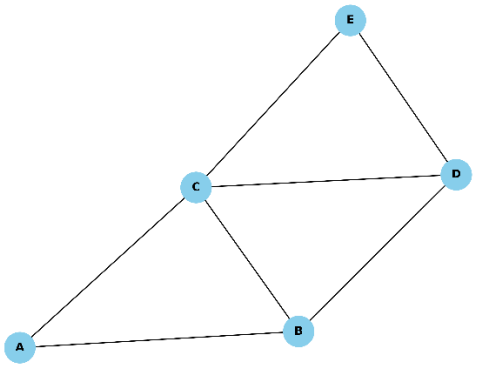
（4）输出参数：

无显式输出参数。结果通过直接访问邻接矩阵数组的地址来实现直接操作。

（5）调用函数：（无）

（6）自测结果：

输入如下的示例图来进行自测：



经过运行测试程序可得输出结果为：

```
Matrix:
0 1 1
1 0 1
1 1 0
```

经判断：

顶点 0 与顶点 1、2 相邻，所以第一行的第二、三列为 1；

顶点 1 与顶点 0、2 相邻，所以第二行的第一、三列为 1；

顶点 2 与顶点 0、1 相邻，所以第三行的第一、二列为 1。

所以测试结果正确，该子程序无误。

附上测试源代码：

```c
#include <stdio.h>
#include <stdlib.h>

// 定义最大顶点数
#define MAX_VERTEX_NUM 100

// 边节点结构体
typedef struct edgenode
{
    int adjvex;               // 邻接点域，存储相邻顶点的下标
    struct edgenode* next;    // 指向下一个邻接点的指针
} edgenode;

// 顶点节点结构体
typedef struct vexnode
{
    edgenode* link;  // 指向第一个邻接点的指针
} vexnode;

// 邻接表转邻接矩阵函数声明
void AdjList2Matrix(vexnode G[], int n, int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]);

int main()
{
    // 示例输入图 G
    vexnode G[3]; // 3 个顶点的图
    // 初始化图 G 的邻接表
    G[0].link = (edgenode*)malloc(sizeof(edgenode));
    G[0].link->adjvex = 1;
    G[0].link->next = (edgenode*)malloc(sizeof(edgenode));
    G[0].link->next->adjvex = 2;
    G[0].link->next->next = NULL;

    G[1].link = (edgenode*)malloc(sizeof(edgenode));
    G[1].link->adjvex = 0;
    G[1].link->next = (edgenode*)malloc(sizeof(edgenode));
```

```
    G[1].link->next->adjvex = 2;
    G[1].link->next->next = NULL;

    G[2].link = (edgenode*)malloc(sizeof(edgenode));
    G[2].link->adjvex = 0;
    G[2].link->next = (edgenode*)malloc(sizeof(edgenode));
    G[2].link->next->adjvex = 1;
    G[2].link->next->next = NULL;

    // 定义邻接矩阵
    int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
    // 调用邻接表转邻接矩阵函数
    AdjList2Matrix(G, 3, Matrix);

    // 打印生成的邻接矩阵
    printf("Matrix:\n");
    for (int i = 0; i < 3; i++) {
        for (int j = 0; j < 3; j++) {
            printf("%d ", Matrix[i][j]);
        }
        printf("\n");
    }

    return 0;
}

// 邻接表转邻接矩阵实现
void AdjList2Matrix(vexnode G[], int n, int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM])
{
    int i, j;
    // 初始化邻接矩阵
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
            Matrix[i][j] = 0;
        }
    }
    // 将邻接表转换为邻接矩阵
    for (i = 0; i < n; i++) {
        edgenode* s = G[i].link;
        while (s != NULL) {
            int temp = s->adjvex;
            Matrix[i][temp] = 1;
            s = s->next;
```

```
        }
    }
}

```

（7）子程序代码：

```
void AdjList2Matrix(vexnode G[], int n, int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM])
{
 int i, j;
 // 初始化邻接矩阵
 for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++) {
   Matrix[i][j] = 0;
  }
 }
// 将邻接表转换为邻接矩阵
for (i = 0; i < n; i++) {
 edgenode* s = G[i].link;
 while (s != NULL) {
  int temp = s->adjvex;
  Matrix[i][temp] = 1;
  s = s->next;
 }
}
}
```

3. 邻接矩阵打印模块：

（1）子程序功能：

打印邻接矩阵，将二维数组中的数据输出到控制台。

（2）子程序编程思路：

使用嵌套的循环遍历二维数组，依次访问每个元素，然后通过 printf 函数将其输出到控制台。

（3）输入参数：

a. Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]：二维数组，存储邻接矩阵的数据；

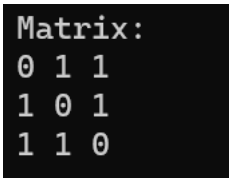
b. n：矩阵的阶数，表示顶点的数量。

（4）输出参数：（无）

（5）调用函数：（无）

（6）自测结果：

上面关于邻接表转邻接矩阵核心模块的已经顺带用过了这个函数片段，即已经测试过了结果无误。



（7）子程序代码：

```
// 打印邻接矩阵
void PrintMatrix(int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM], int n)
{
 printf("Matrix:\n");
 for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
   printf("%d ", Matrix[i][j]);
  }
  printf("\n");
 }
}
```

4. main 主程序内容：

（1）子程序功能：

主程序，用于创建一个无向图的邻接表，然后将该邻接表转换成邻接矩阵，并最后打印输出邻接矩阵。

（2）子程序编程思路：

首先，创建了一个无向图的邻接表 G，其中每个节点 G[i] 都包含了一个顶点和与该顶点相邻的边的信息。

其次，通过循环遍历边的信息，将每条边对应的两个节点插入到邻接表中。

接下来，创建一个邻接矩阵 Matrix，并通过调用 AdjList2Matrix 函数将邻接表转换为邻接矩阵。

最后，通过调用 PrintMatrix 函数，将邻接矩阵输出到控制台。

（3）输入参数：（无）

（4）输出参数: （无）

（5）调用函数:

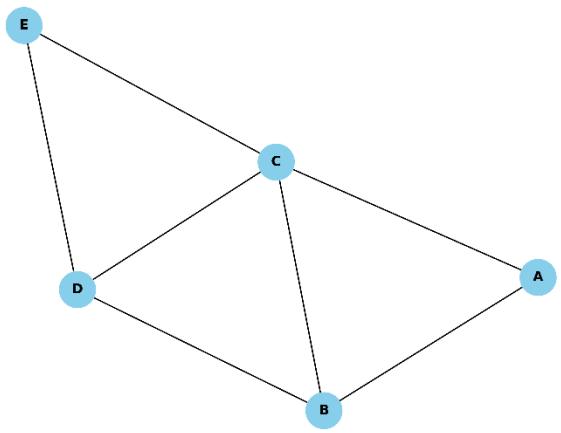
a. AdjList2Matrix: 用于将邻接表转换成邻接矩阵。

b. PrintMatrix: 用于打印邻接矩阵。

（6）自测结果:

由于是 main 主程序，所以我们只需运行整个程序即可进行测试。在 main 主程序中，我预定好的输入为：

这个主程序创建了一个包含 5 个顶点（标记为'A'，'B'，'C'，'D'，'E'）和 6 条边的图。边的连接关系如下：



这是一个无向图，表示通过边连接的顶点关系。下面是每个顶点的邻接表表示：

1.顶点 A 的邻接表：B -> C

2.顶点 B 的邻接表：A ->D

3.顶点 C 的邻接表：A ->D ->E

4.顶点 D 的邻接表：B ->C -> E

5.顶点 E 的邻接表：C ->D

预计输出的正确结果将是图的邻接矩阵。由于这是一个无向图，矩阵是对称的。在这个例子中，部分邻接矩阵的元素值 1，表示相应的顶点之间存在边，如下所示：

A B C D E

A 0 1 1 0 0

B 1 0 0 1 0

C 1 0 0 1 1

D 0 1 1 0 1

E 0 0 1 1 0

这个矩阵表示了每对顶点之间是否有边相连。

而程序经过运行后，控制台得到以下输出：

```
Matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 1 1
0 1 1 0 1
0 0 1 1 0
```

与预期结果一致，所以程序无误。

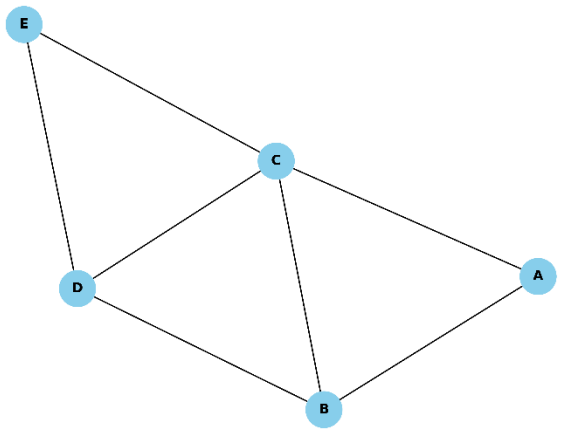
（7）子程序代码:

```
1. int main()
2. {
3.  int n = 5; // 顶点数
4.  int e = 6; // 边数
5.  // 创建邻接表
6.  vexnode G[MAX_VERTEX_NUM];
7.  // 无需输入，直接使用示例图
8.  char vertices[] = { 'A', 'B', 'C', 'D', 'E' };
9.  int edges[][2] = { {0, 1}, {0, 2}, {1, 3}, {2, 3}, {2, 4},
    {3, 4} };
10. // 输入顶点信息
11. for (int k = 0; k < n; k++) {
12.  G[k].vertex = vertices[k];
13.  G[k].link = NULL;
14. }
15. // 输入边信息
16. for (int k = 0; k < e; k++) {
17.  int i = edges[k][0];
18.  int j = edges[k][1];
19.  // 创建一个新的节点，表示边(i, j)
20.  edgenode* s = (edgenode*)malloc(sizeof(edgenode));
21.  s->adjvex = j;
22.  s->next = G[i].link;
23.  G[i].link = s;
24.  // 由于是无向图，还需要创建一个新的节点，表示边(j, i)
```

```
25.   s = (edgenode*)malloc(sizeof(edgenode));
26.   s->adjvex = i;
27.   s->next = G[j].link;
28.   G[j].link = s;
29. }
30. // 创建邻接矩阵
31. int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM];
32. AdjList2Matrix(G, n, Matrix);
33. // 打印邻接矩阵
34. PrintMatrix(Matrix, n);
35.
36. return 0;
37.}
```

三、程序运行结果与说明：

我作为示例输入的图为：



程序经过运行后得到的邻接矩阵如下：

```
Matrix:
0 1 1 0 0
1 0 0 1 0
1 0 0 1 1
0 1 1 0 1
0 0 1 1 0
```

针对上三角元素进行分析可知，矩阵元素(1,2)、(1,3)、(2,4)、(3,4)、(3,5)、(4,5)的值为 1。这在无向图中对应着以下邻接关系：

1. 顶点 A、B、C 互为邻接点；

2. 顶点 B、C、D 互为邻接点；

3.顶点 C、D、E 互为邻接点。

四、所遇问题及解决路径：

1．图的输入和储存：

（1）问题描述：

如何在程序中输入图是一个难题，图自身是一个复杂的、难以量化的数据结构，需要我们取细细分析解决。

（2）解决路径：

将原先的图拆分成边信息和顶点信息，分别使用数组储存。然后将已有的信息通过遍历的形式先将顶点信息写入主表，再将边信息写入侧链，从而构建出图对应的邻接表。

（3）解决效果：

编写代码之后运行结果显示无误，证明了数据的转换传输过程的正确性。

```
int n = 5; // 顶点数
 int e = 6; // 边数
 // 创建邻接表
 vexnode G[MAX_VERTEX_NUM];
 // 无需输入，直接使用示例图
 char vertices[] = { 'A', 'B', 'C', 'D', 'E' };
 int edges[][2] = { {0, 1}, {0, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4} };
 // 输入顶点信息
 for (int k = 0; k < n; k++) {
  G[k].vertex = vertices[k];
  G[k].link = NULL;
 }
 // 输入边信息
 for (int k = 0; k < e; k++) {
  int i = edges[k][0];
  int j = edges[k][1];
  // 创建一个新的节点，表示边(i, j)
  edgenode* s = (edgenode*)malloc(sizeof(edgenode));
  s->adjvex = j;
  s->next = G[i].link;
  G[i].link = s;
  // 由于是无向图，还需要创建一个新的节点，表示边(j, i)
  s = (edgenode*)malloc(sizeof(edgenode));
```

```
24.   s->adjvex = i;
25.   s->next = G[j].link;
26.   G[j].link = s;
27. }
```

四、尚存问题及解决思路提纲：

1.邻接矩阵初始化问题：

目前使用的是最简单粗暴的双循环遍历方法把矩阵中所有元素置为 0，没有找到其他方法，希望能有未知的好方法出现。

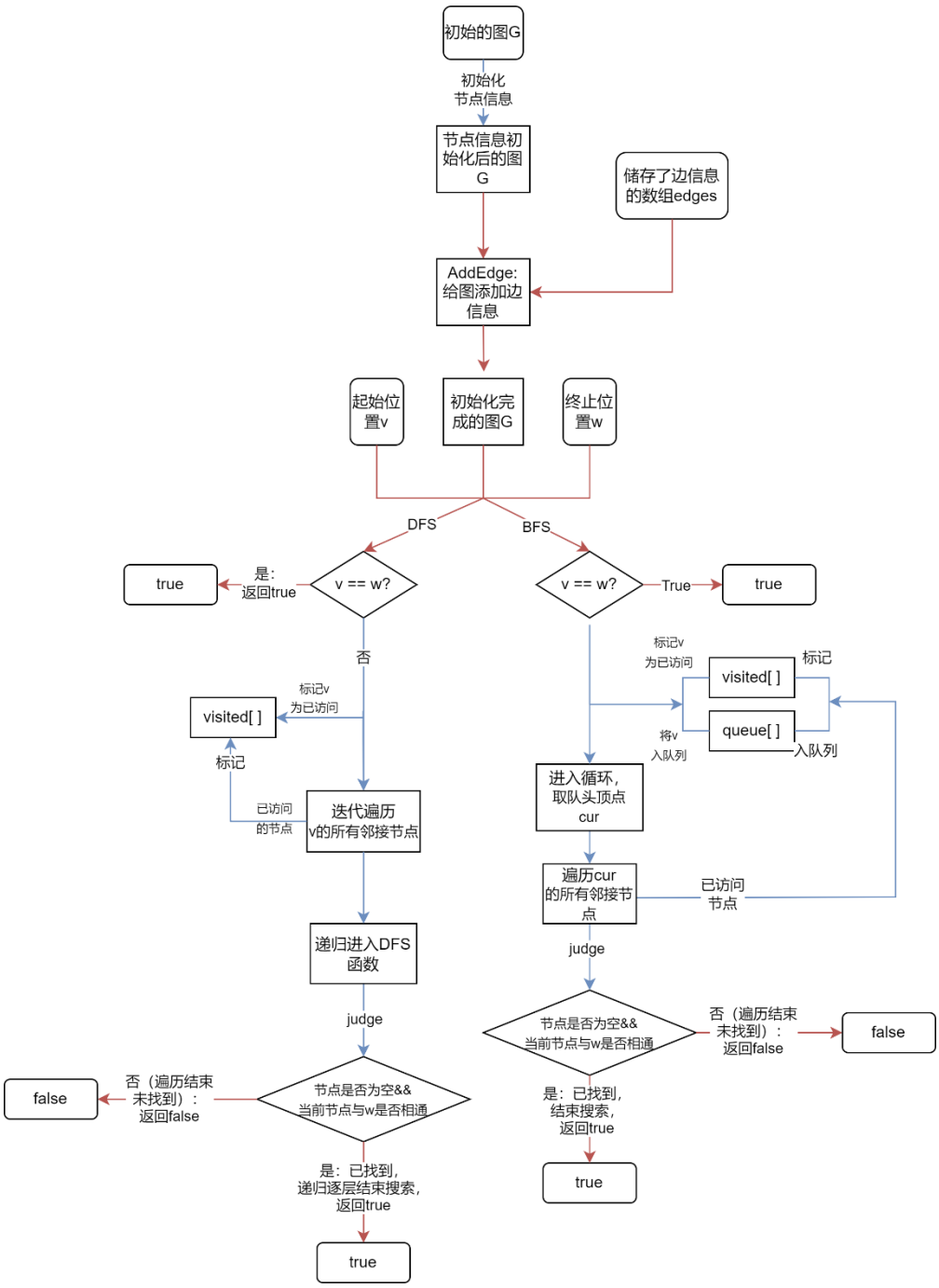
```
1. for (i = 0; i < n; i++) {
2.   for (j = 0; j < n; j++) {
3.    Matrix[i][j] = 0;
4.   }
5.  }
```

2.生成邻接矩阵的函数封装：

在敲代码的过程中，由于种种原因，忘记将图生成邻接矩阵这个较为复杂冗长且重复性强的过程封装在函数中了。图生成邻接矩阵的整个代码流程不算太难，毕竟我都已经整出来了，但却没有封装函数，可见我还需要多多加强函数模块的意识从而提高个人效率。

4. 分别基于 DFS 和 BFS 算法，判断以邻接表存储的有向图是否存在由顶点 v_i 到顶点 v_j 的路径($i \neq j$)。

一、 程序流程图：



二、程序模块简介：

1.图的基本结构的定义模块：

（1）子程序功能：

这段代码定义了图的邻接表表示法，包括顶点结构体、边结构体和邻接表图结构体。它提供了一种灵活的方式来表示图，并支持图的基本操作。

（2）子程序编程思路：

a.定义边结构体：EdgeNode 结构体表示图中的边，包含邻接顶点的下标和指向下一个邻接顶点的指针。

b.定义顶点结构体：VertexNode 结构体表示图中的顶点，包含顶点信息和指向邻接表的头指针。

c.定义图结构体：Graph 结构体表示整个图，包含一个顶点数组和顶点数、边数信息。

（3）输入参数：（无）

（4）输出参数：（无）

（5）调用函数：（无）

（6）自测结果：（无）

（7）子程序代码：

```
1. #include <stdio.h>
2. #include <stdlib.h>
3. #include <stdbool.h>
4.
5. #define MAX_VERTEX_NUM 100
6.
7. // 邻接表中表示边的结构体
8. typedef struct Node
9. {
10.     int adjvex;           // 邻接顶点的下标
11.     struct Node* next;    // 下一个邻接顶点
12.} EdgeNode;
13.
14.// 顶点结构体
15.typedef struct
16.{
17.     char vertex;        // 顶点信息
18.     EdgeNode* link;     // 邻接表头指针
19.} VertexNode;
20.
21.// 邻接表图结构体
22.typedef struct
23.{
```

```
    VertexNode vertices[MAX_VERTEX_NUM];
    int vertexNum;      // 顶点数
    int edgeNum;        // 边数
} Graph;
```

2.有向图初始化模块：

（1）子程序功能：

a. 初始化有向图(initGraph 函数):

初始化有向图的顶点数和边数——为图中的每个顶点创建一个邻接表头，并将其初始化为 NULL。

b. 添加有向边(addDirectedEdge 函数)

在有向图中添加一条从顶点 i 指向顶点 j 的有向边，或者创建一个新的边节点，将其邻接顶点设置为 j 并将新节点插入到顶点 i 对应的邻接表的头部。

（2）子程序编程思路：

a. 初始化有向图(initGraph 函数)：

这个函数主要是设置顶点数和边数，通过遍历每个顶点为其分配顶点信息和初始化邻接表头指针。

b. 添加有向边(addDirectedEdge 函数)

这个函数主要是依据输入的储存边信息的数组给初始的图添加新的边节点。基础的操作主要包含三步：首先将邻接顶点信息设置为 j，再将新节点插入到顶点 i 对应的邻接表的头部，最后更新边数。

（3）输入参数：

a. 初始化有向图(initGraph 函数)

G：指向图结构的指针；

n：顶点数。

b. 添加有向边(addDirectedEdge 函数)

G：指向图结构的指针；

i：起始顶点的下标；

j：终止顶点的下标。

（4）输出参数：（无）

（5）调用函数：（无）

（6）自测结果：（无）

（7）子程序代码：

```
// 初始化有向图
void initGraph(Graph* G, int n)
{
    G->vertexNum = n;
    G->edgeNum = 0;

    // 初始化邻接表
    for (int i = 0; i < n; i++) {
        G->vertices[i].vertex = 'A' + i;
        G->vertices[i].link = NULL;
    }
}

// 添加有向边
void addDirectedEdge(Graph* G, int i, int j)
{
    EdgeNode* newNode = (EdgeNode*)malloc(sizeof(EdgeNode));
    newNode->adjvex = j;
    newNode->next = G->vertices[i].link;
    G->vertices[i].link = newNode;
    G->edgeNum++;
}
```

3.DFS 遍历判断模块：

（1）子程序功能：

通过深度优先搜索(DFS)遍历有向图，判断从顶点 v 是否存在一条路径到达顶点 w。如果找到路径，返回 true，否则返回 false。

（2）子程序编程思路：

在整个过程中，最重要的地方就在于函数的递归调用。首先我们先做出如下判断：如果当前顶点 v 等于目标顶点 w，表示找到路径，返回 true。紧接着便标记当前顶点 v 为已访问。之后进入循环遍历顶点 v 的邻接表，对于每个未访问的邻接顶点，递归调用 DFS。

d. 如果递归调用返回 true，表示找到路径，返回 true。

e. 如果遍历完所有邻接顶点都没有找到路径，则返回 false。

（4）输入参数：

a. G：指向图结构的指针；

b. v：当前顶点的下标；

c. w：目标顶点的下标；

d. visited：记录顶点是否已访问的布尔数组。

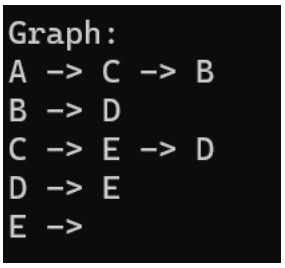
（5）输出参数：

返回布尔值，表示是否存在从顶点 v 到顶点 w 的路径。

（6）调用函数：（无）

（7）自测结果：

输入如下的邻接表，在其中查找 A 和 E 是否相通：



得到的 bool 输出为 true，可见结果正确。

（8）子程序代码：

```
// DFS 遍历判断路径是否存在
bool DFS(Graph* G, int v, int w, bool* visited)
{
    if (v == w) {
        return true; // 找到路径
    }

    visited[v] = true; // 标记当前顶点为已访问

    EdgeNode* p = G->vertices[v].link;
    while (p) {
        if (!visited[p->adjvex] && DFS(G, p->adjvex, w, visited)) {
            return true;
        }
        p = p->next;
    }

    return false;
```

```
19.}
```

4.BFS 遍历判断模块：

（1）子程序功能：

通过广度优先搜索(BFS)遍历有向图，判断从顶点 v 是否存在一条路径到达顶点 w。如果找到路径，返回 true，否则返回 false。同时，我们使用队列数据结构辅助实现 BFS 遍历。

（2）子程序编程思路：

a. 如果起点 v 就是终点 w，直接返回 true;

b. 初始化一个布尔数组 visited 用于记录顶点是否已访问，以及一个整型数组 queue 用于存储待访问的顶点;

c. 将起点 v 标记为已访问，并将其加入队列;

d. 使用队列进行循环，每次出队一个顶点，遍历其邻接顶点;

e. 对于每个未访问的邻接顶点，如果等于目标顶点 w，表示找到路径，返回 true;

f. 否则，标记该邻接顶点为已访问，并将其加入队列;

g. 如果遍历完所有可能的路径都没有找到目标顶点，则返回 false。

（3）输入参数：

a. G：指向图结构的指针;

b. v：起点顶点的下标;

c. w：目标顶点的下标。

（4）输出参数：

返回布尔值，表示是否存在从顶点 v 到顶点 w 的路径。

（5）调用函数：（无）

（6）自测结果：

输入和上面的自测相同的邻接表，在其中查找 A 和 E 是否相通：

```
Graph:
A -> C -> B
B -> D
C -> E -> D
D -> E
E ->
```

得到的 bool 输出为 true，可见结果正确。

（7）子程序代码：

```
// BFS 遍历判断路径是否存在
bool BFS(Graph* G, int v, int w)
{
    if (v == w) {
        return true; // 起点就是终点
    }

    bool visited[MAX_VERTEX_NUM] = { false };
    int queue[MAX_VERTEX_NUM];
    int front = 0, rear = 0;

    visited[v] = true;
    queue[rear++] = v;

    while (front < rear) {
        int cur = queue[front++];

        EdgeNode* p = G->vertices[cur].link;
        while (p) {
            if (!visited[p->adjvex]) {
                if (p->adjvex == w) {
                    return true; // 找到路径
                }
                visited[p->adjvex] = true;
                queue[rear++] = p->adjvex;
            }
            p = p->next;
        }
    }

    return false;
}
```

5.邻接链表打印模块：

（1）子程序功能：

将图的邻接表表示打印出来，以直观展示图的结构。对于每个顶点，输出其邻接顶点的信息。

（2）子程序编程思路：

a. 遍历图中的每个顶点，对于每个顶点，输出其对应的邻接链表;

b. 在每个邻接链表中，遍历顶点的邻接顶点，并输出其相关信息;

c. 使用循环和指针操作遍历数据结构。

（3）输入参数：

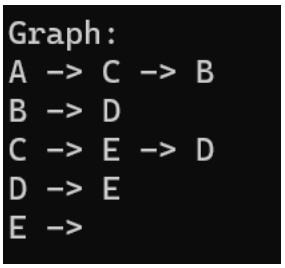
a. G：指向图结构的指针，包含图的顶点和邻接表信息。

（4）输出参数：（无）

（5）调用函数：（无）

（6）自测结果：

输入和上面的自测相同的图 G 的邻接表将其打印出来：



打印结果正确无误，因此该代码片段正确。

（7）子程序代码：

```
1. // 打印邻接表
2. void printGraph(Graph* G)
3. {
4.     printf("Graph:\n");
5.     for (int i = 0; i < G->vertexNum; i++) {
6.         printf("%c -> ", G->vertices[i].vertex);
7.         EdgeNode* p = G->vertices[i].link;
8.         while (p) {
9.             if (p->next != NULL) {
10.                 printf("%c -> ", G->vertices[p->adjvex].ver
   tex);
11.             }
12.             else {
13.                 printf("%c", G->vertices[p->adjvex].vertex)
   ;
14.             }
15.
16.             p = p->next;
17.         }
18.         printf("\n");
19.     }
```

```
20.}
```

5. main 主程序模块：

（1）子程序功能：

首先，初始化图(initGraph 函数)——为图结构分配内存，并设置顶点数。之后，添加有向边(addDirectedEdge 函数)。接着，打印邻接表(printGraph 函数)，将图的邻接表表示打印出来，以直观展示图的结构。

然后，分别使用深度优先搜索(DFS 函数)和广度优先搜索(BFS 函数)，判断图中是否存在从指定起点到终点的路径，并将结果打印出来。

（2）子程序编程思路：

首先，我们初始化图 (调用 initGraph 函数)：分配内存给图结构，初始化顶点数和邻接表;之后添加有向边 (调用 addDirectedEdge 函数):在有向图中，将边添加到起始顶点的邻接链表中并打印邻接表(调用 printGraph 函数)。

之后我们首先通过深度优先搜索(DFS 函数):使用递归实现深度优先搜索，标记已访问的顶点，查找路径是否存在;广度优先搜索 (BFS 函数):使用队列实现广度优先搜索，标记已访问的顶点，查找路径是否存在。

（3）输入参数：

a. initGraph 函数：G - 指向图结构的指针，n - 顶点数。

b. addDirectedEdge 函数：G - 指向图结构的指针，start - 边的起始顶点，end - 边的终点。

c. printGraph 函数：G - 指向图结构的指针。

d. DFS 函数：G - 指向图结构的指针，start - 深度优先搜索的起点，end - 深度优先搜索的终点，visited - 用于记录已访问顶点的数组。

e. BFS 函数：G - 指向图结构的指针，start - 广度优先搜索的起点，end - 广度优先搜索的终点。

（4）输出参数：

a. initGraph 函数：（无）

b. addDirectedEdge 函数：（无）

c. printGraph 函数：（无）

d. DFS 函数：布尔值，表示路径是否存在；

e. BFS 函数：布尔值，表示路径是否存在。

（5）调用函数：

在 main 中调用了 initGraph, addDirectedEdge, printGraph, DFS 和 BFS 函数。

（6）自测结果：

输入和上面的自测相同的邻接表，在其中查找 A 和 E 是否相通：

```
Graph:
A -> C -> B
B -> D
C -> E -> D
D -> E
E ->
```

结果如下：

```
DFS: Path from A to E exists
BFS: Path from A to E exists
```

可知结果无误，程序正确。

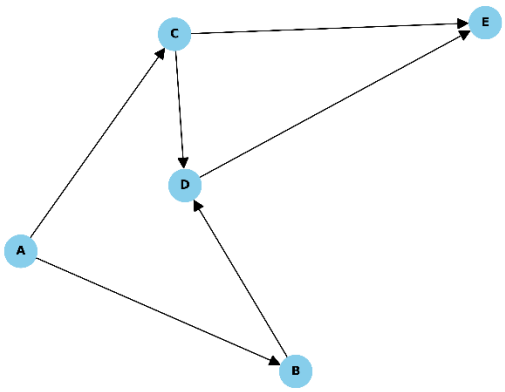
（7）子程序代码：

```
1. int main()
2. {
3.     Graph G;
4.     int n = 5; // 顶点数
5.     initGraph(&G, n);
6.
7.     // 直接使用示例图
8.     int edges[][2] = { {0, 1}, {0, 2}, {1, 3}, {2, 3}, {2,
   4}, {3, 4} };
9.     for (int k = 0; k < 6; k++) {
10.        addDirectedEdge(&G, edges[k][0], edges[k][1]);
11.    }
12.
13.    // 打印图的邻接表
14.    printGraph(&G);
15.
16.    // 判断路径是否存在
17.    int startVertex = 0; // 起点为顶点A
18.    int endVertex = 4;   // 终点为顶点E
19.
```

```
20.     printf("\nDFS: Path from %c to %c %s\n", G.vertices[sta
    rtVertex].vertex, G.vertices[endVertex].vertex,
21.         DFS(&G, startVertex, endVertex, (bool*)calloc(n, si
    zeof(bool))) ? "exists" : "does not exist");
22.
23.     printf("BFS: Path from %c to %c %s\n", G.vertices[start
    Vertex].vertex, G.vertices[endVertex].vertex,
24.         BFS(&G, startVertex, endVertex) ? "exists" : "does
    not exist");
25.
26.     return 0;
27. }
```

三、程序运行结果及说明：

在该程序中我们输入的示例图的逻辑结构如下所示，而我们需要在其中查找 A 和 E 是否相通，易得 A 和 E 肯定相通：



该图对应的邻接表如下所示：

```
Graph:
A -> C -> B
B -> D
C -> E -> D
D -> E
E ->
```

得到的运行结果如下图所示：

```
DFS: Path from A to E exists
BFS: Path from A to E exists
```

可知结果无误，程序正确。

三、所遇问题及解决途径：

（暂无）

四、尚存问题及解决思路提纲：

1.示例过于简单：

（1）问题描述：

在这题中我使用的示例图较为简单，虽称为有向，但其实点与点之间都为单向联系而非双向。算法的普适性不强，可能会出现问题。

（2）解决思路：

我觉得我的算法没有什么大问题，暂时自己想不出来会有什么问题，这只是一种担心罢了。

2.搜索之后未返回路径信息：

（1）问题描述：

在这题中，我没有在函数中设置返回路径的具体信息而只是简单的返回了布尔值来显示路径是否存在。这样的话功能是不完全的。

（2）解决思路：

个人主要想的是定义一个结构体 PathInfo，中间储存两个变量：一个布尔值显示路径是否存在，而另一个链表则用来储存路径的具体信息。虽然并没有做出具体的修改（毕竟题干中只要求判断是不是而非是什么，而且个人感觉有些累不想改了），但是这个结构体 PathInfo 我感觉很重要，在未来一定会起作用的。

~~2.两种算法写在了一个程序里面：~~

~~（1）问题描述：~~

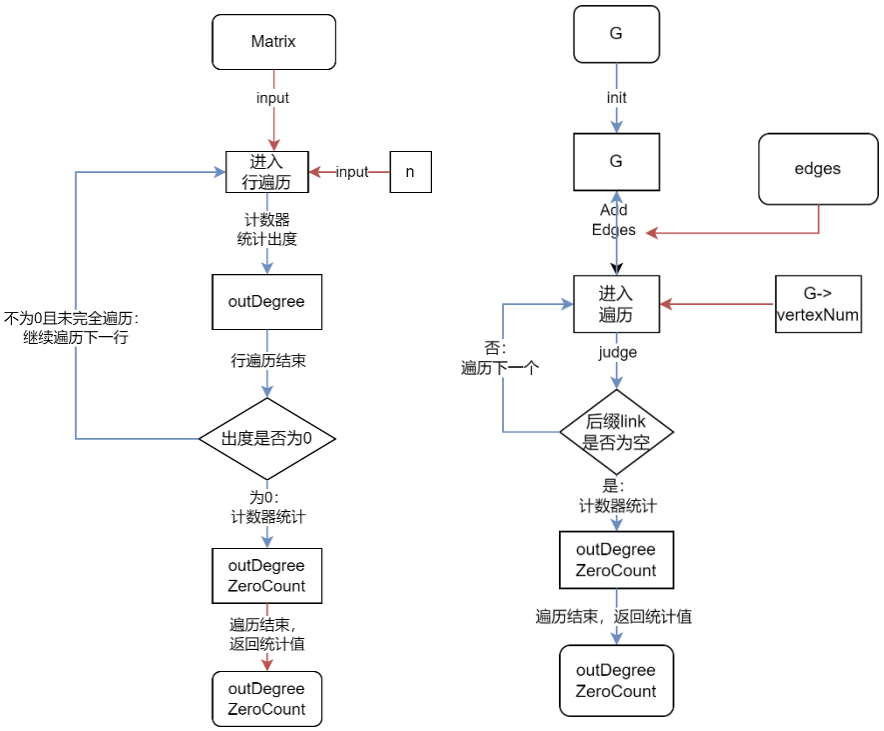
~~按照常理来说，DFS 和 BFS 算法写的两个程序应当各成一派，结果我给人家撮合到一块儿去了。~~

~~（2）解决思路：~~

~~分开也很简单,其他部分 copy 一份就行了，计算机比现实方便多了。不过懒得搞——好不容易合在一块的为什么又要分了？~~

5. 设G为一个有n个顶点的有向图，分别以邻接矩阵和邻接表作为存储结构，设计计算有向图G出度为 0 的顶点个数的算法。

一、 程序流程图：



二、程序模块简介：

邻接矩阵：

1. 计算有向图 G 出度为 0 的顶点个数（邻接矩阵）：

(1) 子程序功能：

该子程序的功能是计算以邻接矩阵为存储格式的有向图 G 中出度为 0 的顶点个数。

(2) 子程序编程思路：

首先初始化计数器 outDegreeZeroCount 为 0；之后进入两层循环遍历邻接矩阵中的每个顶点并判断内层循环计算当前顶点的出度，即邻接矩阵中每一行的非零元素的个数——如果出度为 0，则将 outDegreeZeroCount 增加 1；最终遍历结束之后返回最终的出度为 0 的顶点个数。

(3) 输入参数：

a. Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]：表示有向图 G 的邻接矩阵；

b. n：表示图 G 的顶点数目。

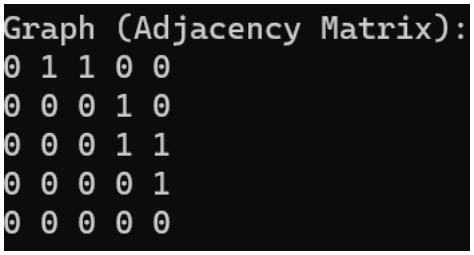
(4) 输出参数：

a. outDegreeZeroCount：表示有向图 G 中出度为 0 的顶点个数。

(5) 调用函数：（无）

(6) 自测结果：

输入如下的邻接矩阵：



得到的返回值为 1，可见结果正确程序无误。

(7) 子程序代码：

```
1.    // 计算有向图G出度为0的顶点个数（邻接矩阵）
2. int calculateOutDegreeZeroInAdjMatrix(int Matrix[MAX_VERTEX
   _NUM][MAX_VERTEX_NUM], int n)
3. {
4.     int outDegreeZeroCount = 0;
5.
6.     // 遍历每个顶点，检查其出度是否为0
7.     for (int i = 0; i < n; i++) {
8.         int outDegree = 0;
9.         for (int j = 0; j < n; j++) {
10.            if (Matrix[i][j] != 0) {
11.                outDegree++;
12.            }
13.        }
14.        if (outDegree == 0) {
15.            outDegreeZeroCount++;
16.        }
17.    }
18.
19.    return outDegreeZeroCount;
20.}
21.
```

2. 邻接矩阵打印模块：

(1) 子程序功能：

该子程序的功能是打印图的邻接矩阵。

(2) 子程序编程思路：

首先使用嵌套循环遍历邻接矩阵中的每个元素，而对于每个元素则使用 printf 函数将其值输出到屏幕上;在每行结束后，通过 printf("\n");换行，以保持矩阵的行列结构。

(3) 输入参数：

a. Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM]：表示图的邻接矩阵；

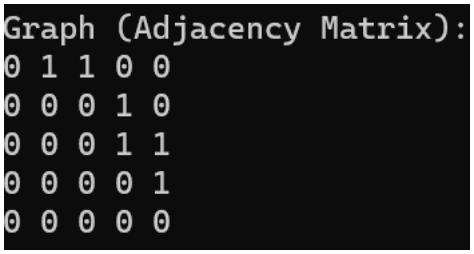
b. n：表示图的顶点数目。

(4) 输出参数：（无）

(5) 调用函数：（无）

(6) 自测结果：

该函数打印邻接矩阵的结果如下：



可知代码正确无误。

(7) 子程序代码：

```
1.   // 打印邻接矩阵
2. void printGraphAdjMatrix(int Matrix[MAX_VERTEX_NUM][MAX_VER
   TEX_NUM], int n)
3. {
4.     printf("Graph (Adjacency Matrix):\n");
5.     for (int i = 0; i < n; i++) {
6.         for (int j = 0; j < n; j++) {
7.             printf("%d ", Matrix[i][j]);
8.         }
9.         printf("\n");
10.    }
11.}
```

3.main 主程序：

(1) 子程序功能：

main 主程序是程序的入口，它包含了示例有向图的邻接矩阵表示，打印邻接矩阵图，以及调用函数计算邻接矩阵图的出度为 0 的顶点个数并输出结果。

(2) 子程序编程思路：

首先我们创建一个有向图的邻接矩阵表示并随后调用 printGraphAdjMatrix 函数，打印图的邻接矩阵；接着我们调用 calculateOutDegreeZeroInAdjMatrix 函数计算出度为 0 的顶点个数并将结果打印出来。

(3) 输入参数：（无）

(4) 输出参数：（无）

(5) 调用函数：

a. printGraphAdjMatrix(Matrix, 5)：打印图的邻接矩阵;

b. calculateOutDegreeZeroInAdjMatrix(Matrix, 5)：计算邻接矩阵图的出度为 0 的顶点个数。

(6) 自测结果：

程序以下面的邻接矩阵作为输入：

```
Graph (Adjacency Matrix):
0 1 1 0 0
0 0 0 1 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
```

得到的输出结果为：

```
Number of vertices with out-degree zero (Adjacency Matrix): 1
```

可知程序正确无误。

(7) 子程序代码：

```
1. int main()
2. {
3.     // 示例有向图的邻接矩阵表示
4.     int Matrix[MAX_VERTEX_NUM][MAX_VERTEX_NUM] = {
5.         {0, 1, 1, 0, 0},
6.         {0, 0, 0, 1, 0},
7.         {0, 0, 0, 1, 1},
8.         {0, 0, 0, 0, 1},
```

```
        {0, 0, 0, 0, 0}
    };

    // 打印邻接矩阵图
    printGraphAdjMatrix(Matrix, 5);

    // 计算邻接矩阵图的出度为 0 的顶点个数
    int outDegreeZeroCount_Matrix = calculateOutDegreeZeroInAdjMatrix(Matrix, 5);
    printf("\nNumber of vertices with out-degree zero (Adjacency Matrix): %d\n", outDegreeZeroCount_Matrix);

    return 0;
}
```

邻接表：

1.数据结构初始化模块：

(1)子程序功能：

该代码片段定义了以邻接表为存储结构的图表示，包括边的结构体 EdgeNode，顶点的结构体 VertexNode，以及图的结构体 Graph。

(2)子程序编程思路：

邻接表是一种图的表示方法，其中每个顶点的邻接关系通过链表来表示。EdgeNode 结构体表示图中的边，包含邻接顶点的下标和指向下一个邻接顶点的指针。VertexNode 结构体表示图中的顶点，包含顶点信息和指向邻接表头的指针。Graph 结构体包含顶点数组，记录图的顶点数和边数。

(3)输入参数：（无）

(4)输出参数：（无）

(5)调用函数：（无）

(6)自测结果：（无）

(7)子程序代码：

```
// 以邻接表为存储结构
#include <stdio.h>
#include <stdlib.h>

```

```
#define MAX_VERTEX_NUM 100

// 邻接表中表示边的结构体
typedef struct Node
{
    int adjvex;           // 邻接顶点的下标
    struct Node* next;    // 下一个邻接顶点
} EdgeNode;

// 顶点结构体
typedef struct
{
    char vertex;        // 顶点信息
    EdgeNode* link;     // 邻接表头指针
} VertexNode;

// 邻接表图结构体
typedef struct
{
    VertexNode vertices[MAX_VERTEX_NUM];
    int vertexNum;      // 顶点数
    int edgeNum;        // 边数
} Graph;
```

2.计算有向图 G 出度为 0 的顶点个数模块：

(1)子程序功能：

该子程序的功能是计算有向图 G 中出度为 0 的顶点个数。

(2)子程序编程思路：

遍历图中的每个顶点，检查其邻接表头指针是否为空。如果为空，表示该顶点的出度为 0，计数器递增。

(3)输入参数：

a. Graph* G：有向图的指针，通过邻接表表示。

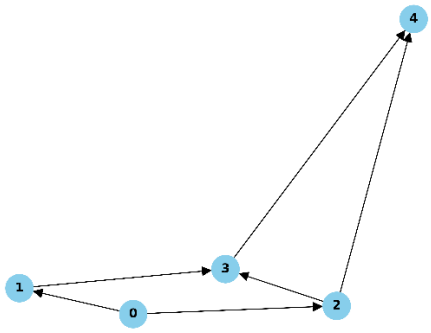
(4) 输出参数：

a. outDegreeZeroCount:返回有向图 G 中出度为 0 的顶点个数。

(5) 调用函数：（无）

(6) 自测结果：

输入如下的邻接表对应的图：




```
Graph (Adjacency List):
A -> C -> B
B -> D
C -> E -> D
D -> E
E ->
```

得到的返回值为 1，可见代码的正确性。

(7) 子程序代码：

```
1. // 计算有向图G 出度为0 的顶点个数（邻接表）
2. int calculateOutDegreeZeroInAdjList(Graph* G)
3. {
4.     int outDegreeZeroCount = 0;
5.
6.     // 遍历每个顶点，检查其出度是否为0
7.     for (int i = 0; i < G->vertexNum; i++) {
8.         EdgeNode* p = G->vertices[i].link;
9.         if (p == NULL) {
10.            outDegreeZeroCount++;
11.        }
12.    }
13.
14.    return outDegreeZeroCount;
15.}
```

3.邻接表打印模块：

(1)子程序功能：

该子程序的功能是打印有向图 G 的邻接表。

(2)子程序编程思路：

使用循环遍历图中的每个顶点，对每个顶点，打印其顶点信息，然后遍历其邻接表，打印邻接点的信息。

(3)输入参数：

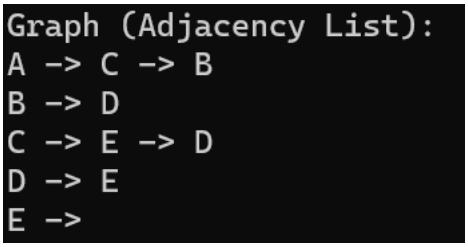
a. Graph* G：有向图的指针，通过邻接表表示。

(4) 输出参数：（无）

(5) 调用函数：（无）

(6) 自测结果：

运行该函数打印邻接表的结果如下：



由此可知代码的正确性。

(7)子程序代码：

```
1. // 打印邻接表
2. void printGraphAdjList(Graph* G)
3. {
4.     printf("Graph (Adjacency List):\n");
5.     for (int i = 0; i < G->vertexNum; i++) {
6.         printf("%c -> ", G->vertices[i].vertex);
7.         EdgeNode* p = G->vertices[i].link;
8.         while (p) {
9.             if (p->next != NULL) {
10.                 printf("%c -> ", G->vertices[p->adjvex].ver
   tex);
11.             }
12.             else {
13.                 printf("%c", G->vertices[p->adjvex].vertex)
   ;
14.             }
15.             p = p->next;
16.         }
17.         printf("\n");
18.     }
19.}
```

4.main 主程序：

（1）子程序功能：

创建一个有向图的邻接表表示，并且对其进行打印，接着调用函数计算出度为 0 的顶点的数量并将结果打印出来。

（2）子程序编程思路：

首先，初始化邻接表图：创建一个表示有向图的邻接表结构，并初始化每个顶点以及它们的邻接表；接着，创建示例有向图的邻接表：使用示例的边数组，在邻接表中添加边并打印邻接表图：使用 printGraphAdjList()函数打印图的邻接表。之后，计算出度为 0 的顶点个数：使用 calculateOutDegreeZeroInAdjList()函数计算出度为 0 的顶点数量并将结果打印出来。

（3）输入参数：（无）

（4）输出参数：（无）

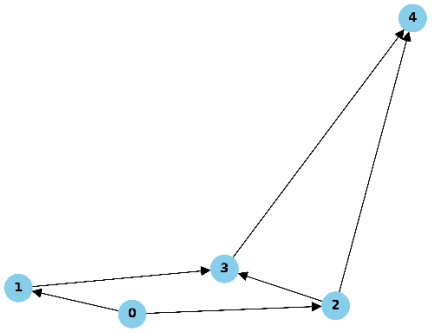
（5）调用函数：

a. printGraphAdjList(&G_adjList): 用于打印图的邻接表；

b. calculateOutDegreeZeroInAdjList(&G_adjList): 用于计算邻接表图中出度为 0 的顶点数量。

（6）自测结果：

选取的示例图如下：



得到的输出的邻接表如下：

```
Graph (Adjacency List):
A -> C -> B
B -> D
C -> E -> D
D -> E
E ->
```

得到的输出的最终结果如下：

```
Number of vertices with out-degree zero (Adjacency List): 1
```

由此可见代码的正确性。

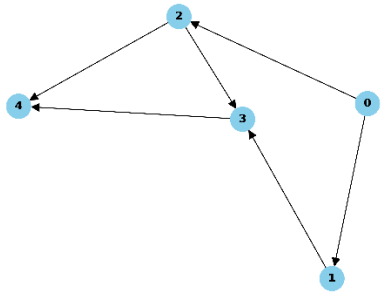
（7）子程序代码：

```
int main()
{
    // 初始化邻接表图
    Graph G_adjList;
    G_adjList.vertexNum = 5; // 顶点数
    G_adjList.edgeNum = 0;

    // 初始化邻接表
    for (int i = 0; i < 5; i++) {
        G_adjList.vertices[i].vertex = 'A' + i;
        G_adjList.vertices[i].link = NULL;
    }

    // 示例有向图的邻接表表示
    int edges_adjList[][2] = { {0, 1}, {0, 2}, {1, 3}, {2, 3}, {2, 4}, {3, 4} };
    for (int k = 0; k < 6; k++) {
        EdgeNode* newNode = (EdgeNode*)malloc(sizeof(EdgeNode));
        newNode->adjvex = edges_adjList[k][1];
        newNode->next = G_adjList.vertices[edges_adjList[k][0]].link;
        G_adjList.vertices[edges_adjList[k][0]].link = newNode;
        G_adjList.edgeNum++;
    }

    // 打印邻接表图
    printGraphAdjList(&G_adjList);

    // 计算邻接表图的出度为0 的顶点个数
    int outDegreeZeroCount_adjList = calculateOutDegreeZeroInAdjList(&G_adjList);
    printf("\nNumber of vertices with out-degree zero (Adjacency List): %d\n", outDegreeZeroCount_adjList);

    return 0;
}
```

三、程序运行结果及说明：

邻接矩阵：

我选取的示例图的逻辑结构如下，易得该图出度为 0 的节点只有 1 个：



其对应的邻接矩阵为：

```
Graph (Adjacency Matrix):
0 1 1 0 0
0 0 0 1 0
0 0 0 1 1
0 0 0 0 1
0 0 0 0 0
```

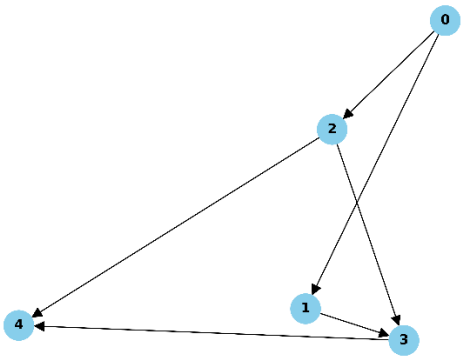
程序运行的结果为：

```
Number of vertices with out-degree zero (Adjacency Matrix): 1
```

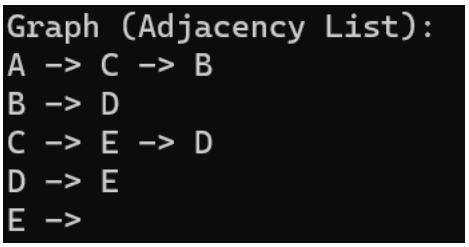
由此可知程序运行结果正确。

邻接表：

我选取的示例图的逻辑结构如下，易得该图出度为 0 的节点只有 1 个：



其对应的邻接表为：



得到的输出的最终结果如下：

由此可知程序运行结果正确。

三、所遇问题及解决途径：

（暂无）

四、尚存问题及解决思路提纲：

1.示例过于简单：

（1）问题描述：

在这题中我使用的示例图较为简单，虽称为有向，但其实点与点之间都为单向联系而非双向。算法的普适性不强，可能会出现问题，这或许只是一种担心罢了。

（2）解决思路：

我认为：对于求解出度为 0 的节点数量的实现关键在于考虑出边的存在与否，而不考虑双向和单向。即使是双向图，该算法也是依旧正确的。