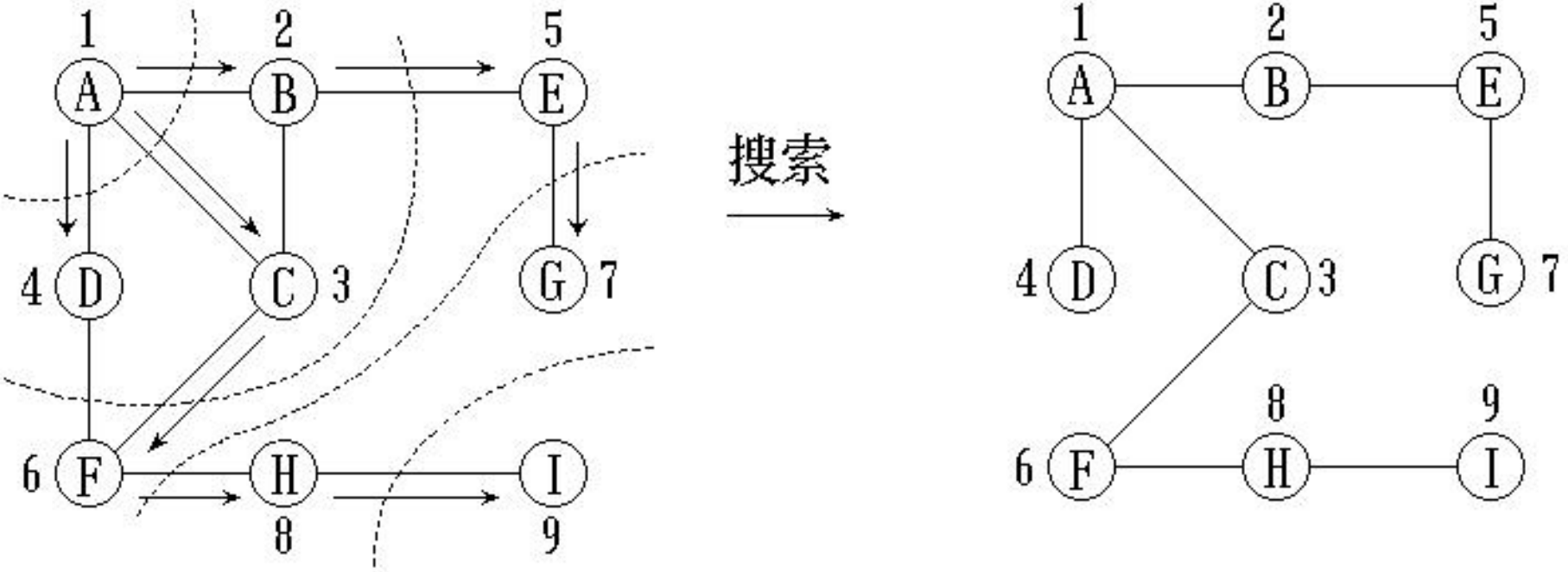
四 、图的搜索算法

4.3 图的搜索算法

- 广度优先搜索BFS (Breadth First Search)
  - 首先访问指定顶点v，将v作为当前顶点
  - 访问当前顶点的所有未访问过的邻接点，并依次将访问的这些邻接点作为当前顶点
  - 重复2， 直到所有顶点被访问为止

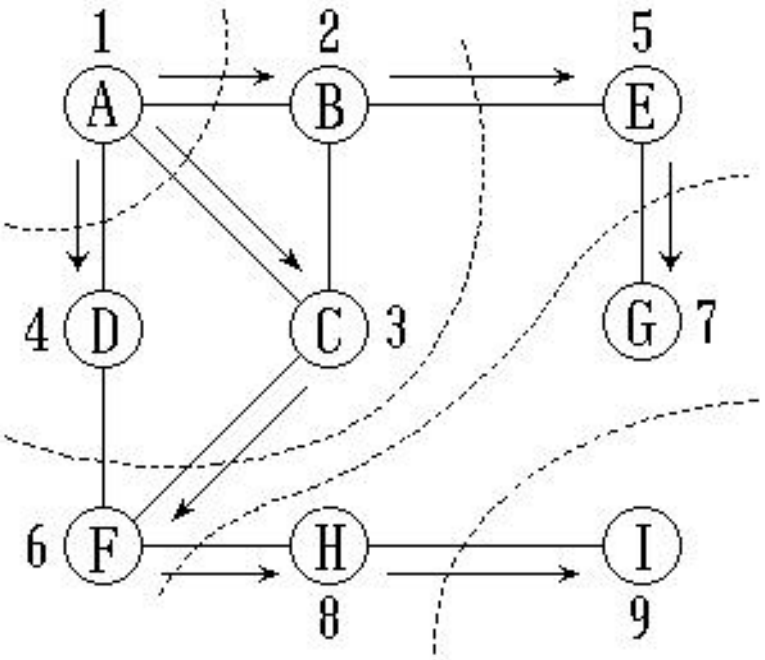
4.3 图的搜索算法

- 广度优先搜索BFS (Breadth First Search)



4.3 图的搜索算法

- 邻接矩阵



	A	B	C	D	E	F	G	H	I
A	0	1	1	1	0	0	0	0	0
B	1	0	1	0	1	0	0	0	0
C	1	1	0	0	0	1	0	0	0
D	1	0	0	0	0	1	0	0	0
E	0	1	0	0	0	0	1	0	0
F	0	0	1	1	0	0	0	1	0
G	0	0	0	0	1	0	0	0	0
H	0	0	0	0	0	1	0	0	1
I	0	0	0	0	0	0	0	1	0

4.3 图的搜索算法

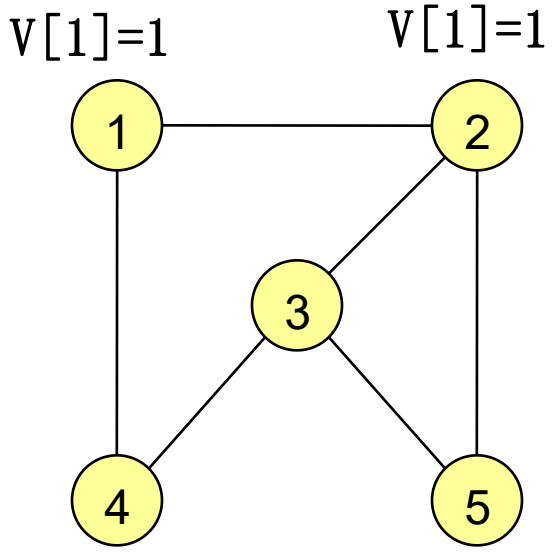
- 广度优先搜索算法

```
int g[100][100]={……}; //初始化邻接矩阵
int n;  //图中顶点个数
bfs(int k)
{
   int i,j;
   InitQueue(Q);  //初始化一个队列
   printf("搜索到%d\n",k);
   visited[k]=1;
   EnQueue(Q,k);  //k顶点进队列
```

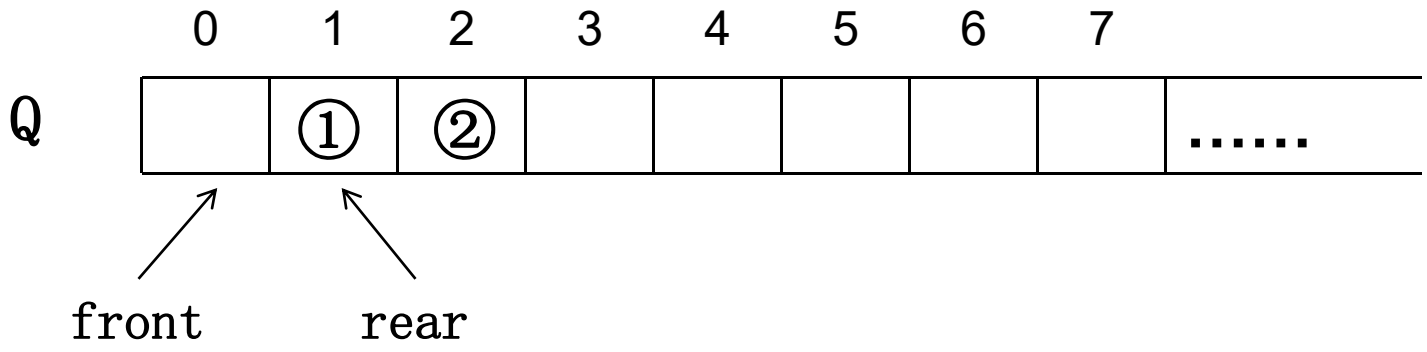
4.3 图的搜索算法

- 广度优先搜索算法

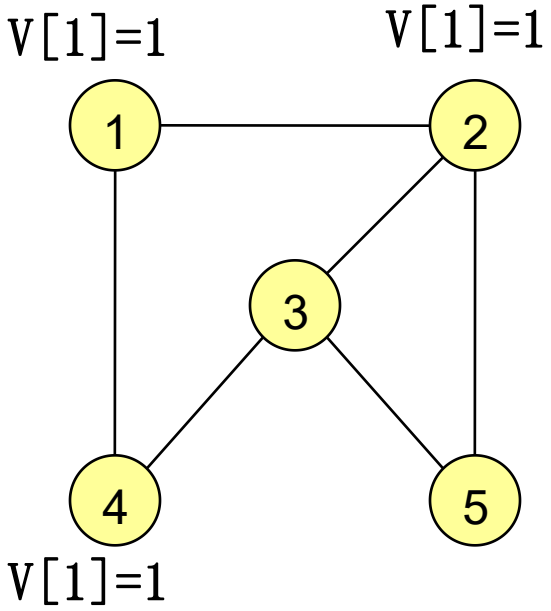
```
while(!QueueEmpty(Q))
{
    i=DeQueue(Q);    //出队列给i
    for(j=1;j<=n;j++)  //搜索邻接矩阵中i行每个元素
        if(g[i][j]==1&&visited[j]==0) //判断1——n顶点与i顶点是否邻
                                      接边，如果是且未被访问过
        {
            printf("搜索到%d\n",j);  //输出i顶点的邻接点
            visited[j]=1;
            EnQueue(Q,j);  //把该邻接点进队列
        }
}
}
```



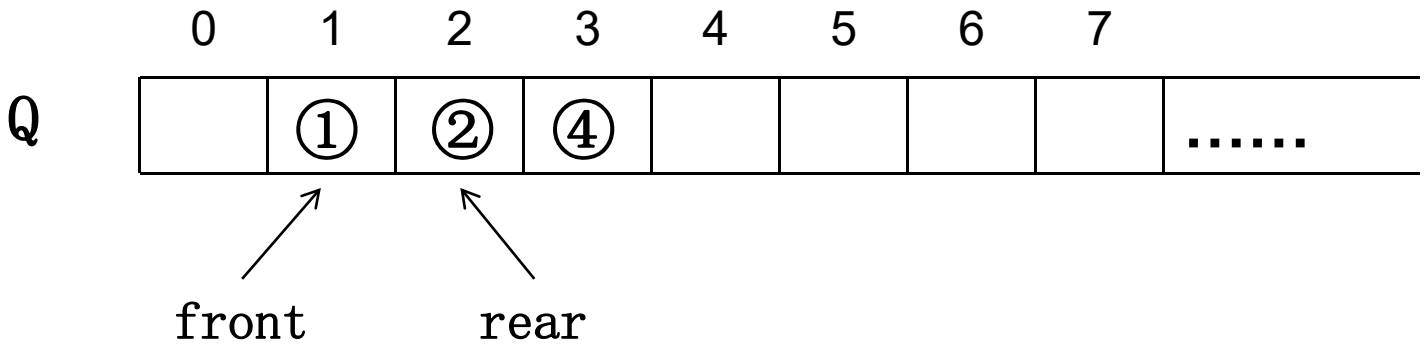
	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0



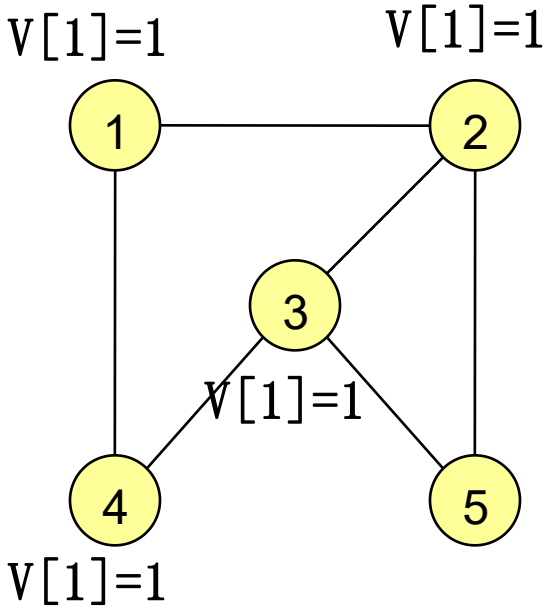
输出：① ②



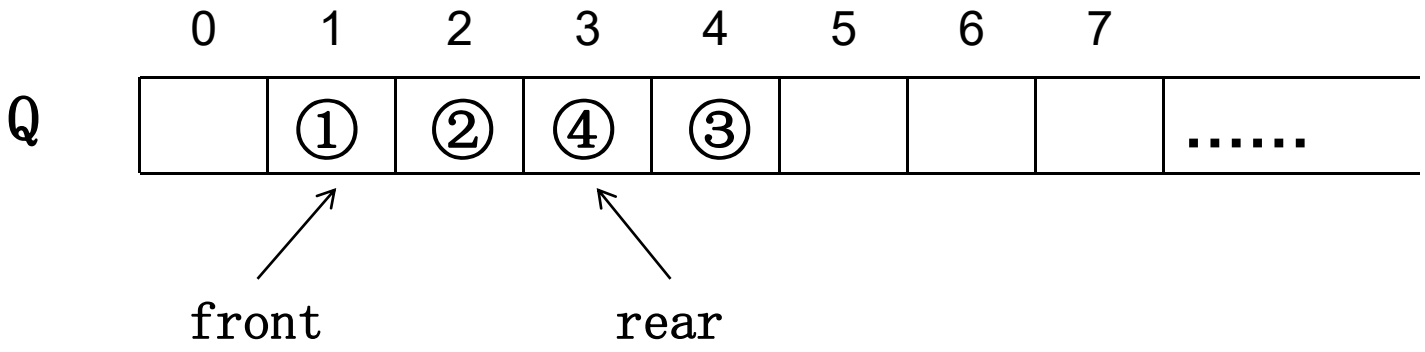
	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0



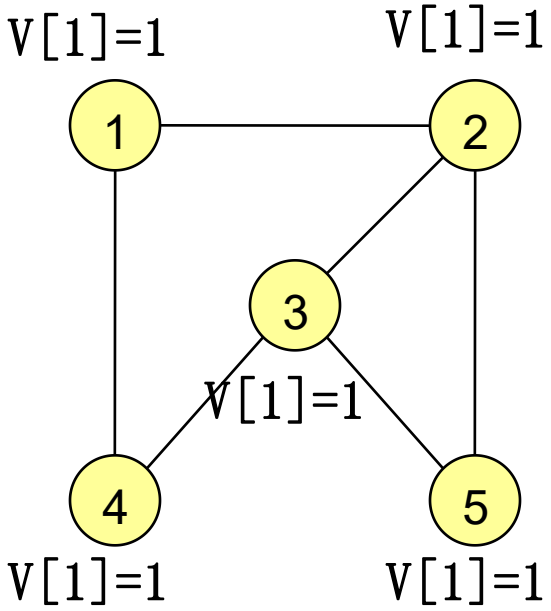
输出：① ② ④



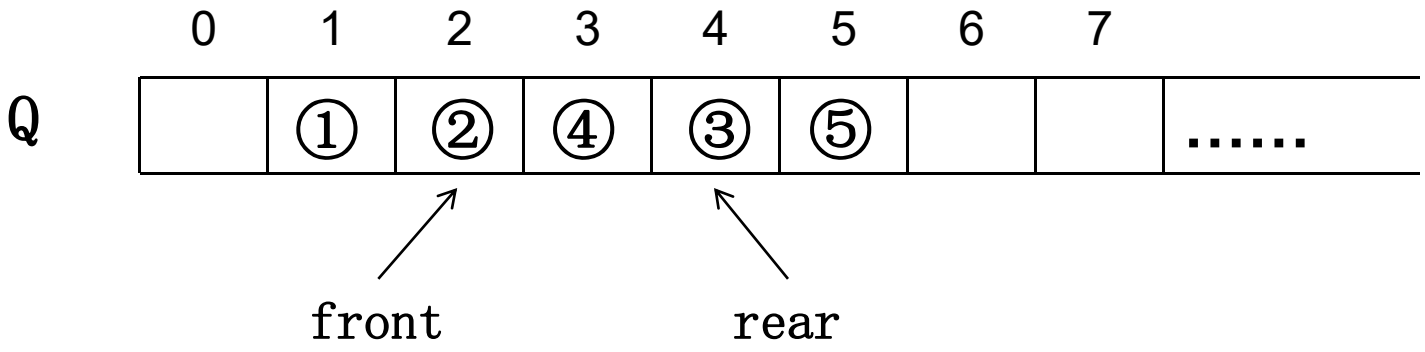
	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0



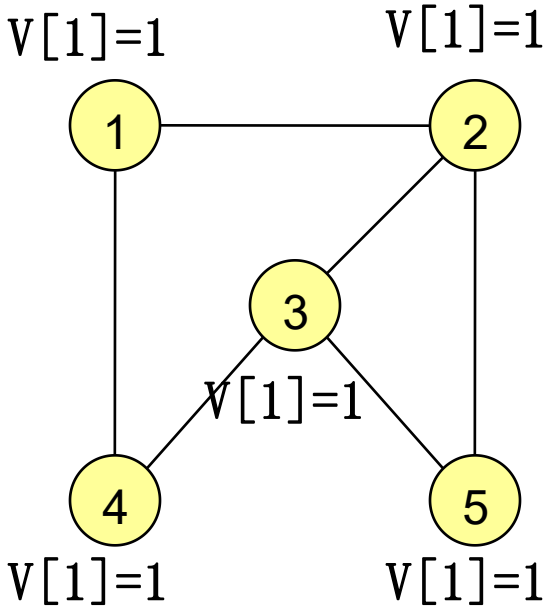
输出：① ② ④ ③



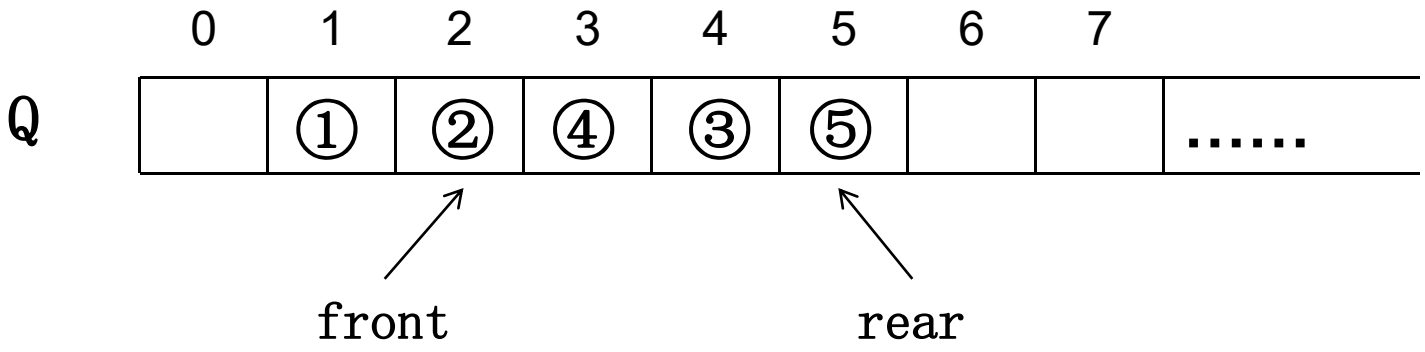
	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0



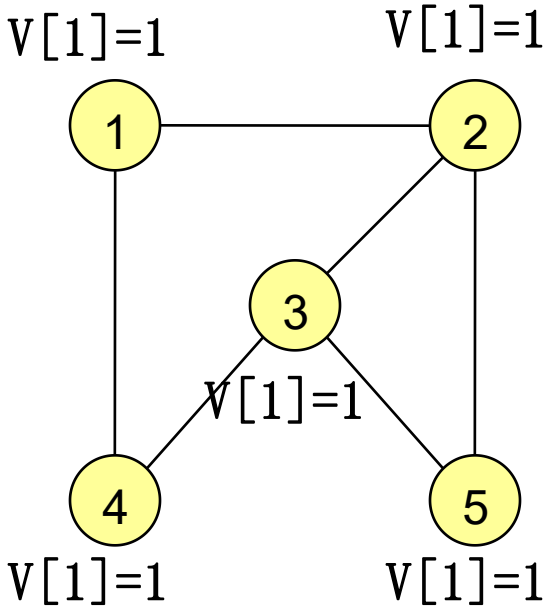
输出：① ② ④ ③ ⑤



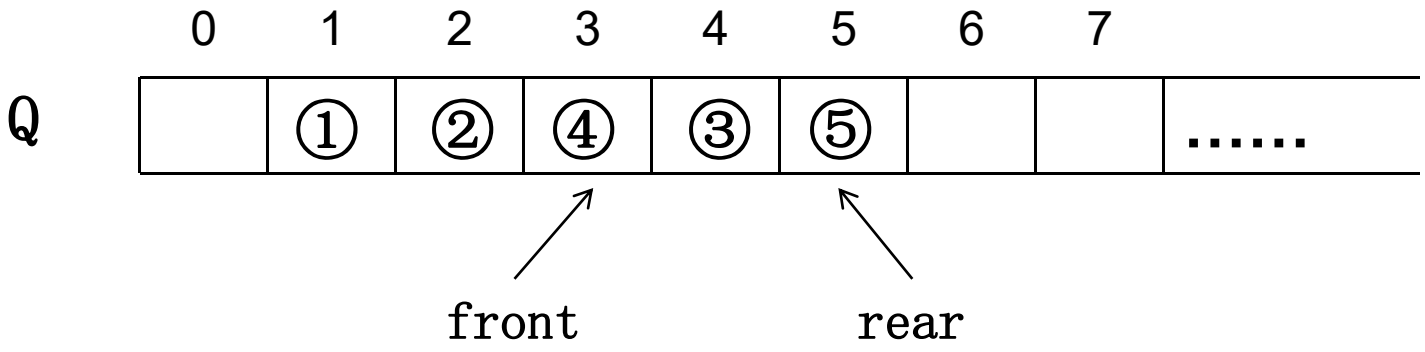
	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0



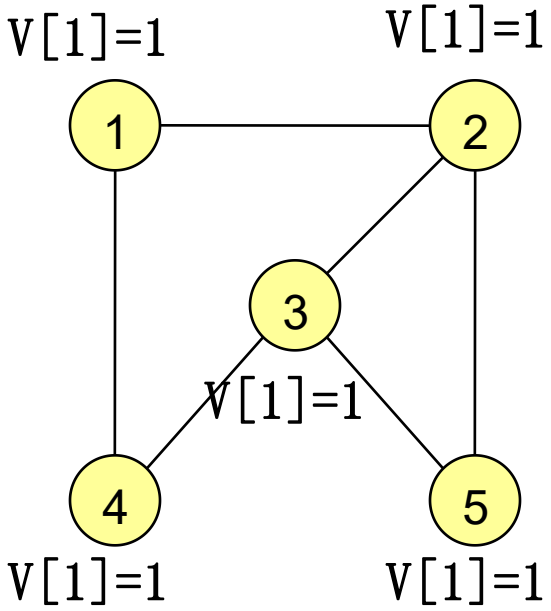
输出：① ② ④ ③ ⑤



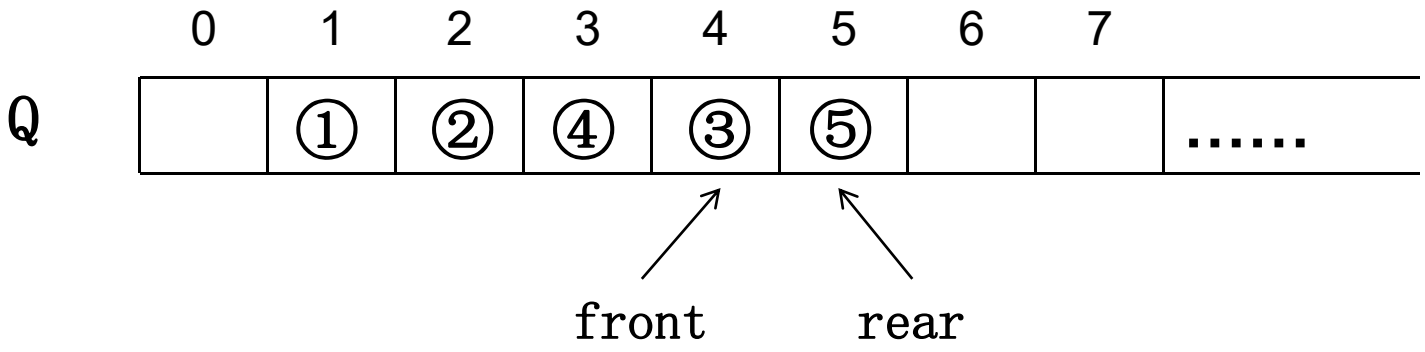
	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0



输出：① ② ④ ③ ⑤



	①	②	③	④	⑤
①	0	1	0	1	0
②	1	0	1	0	1
③	0	1	0	1	1
④	1	0	1	0	0
⑤	0	1	1	0	0



输出：① ② ④ ③ ⑤

4.3 图的搜索算法

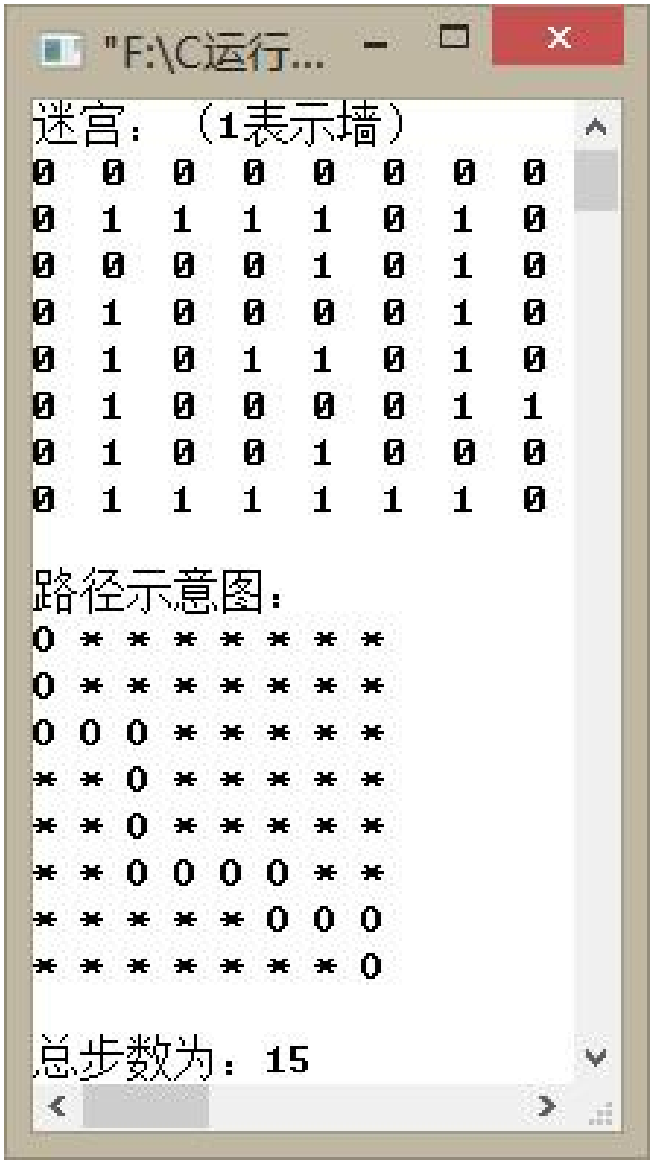
- 例：走迷宫问题（BFS）
  - 迷宫是许多小方格构成的矩形，在每个小方格中有的是墙（1）有的是路（0）。设迷宫的入口是在左上角(1,1)，出口是右下角(8,8)。根据给定的迷宫，找出一条从入口到出口的路径。

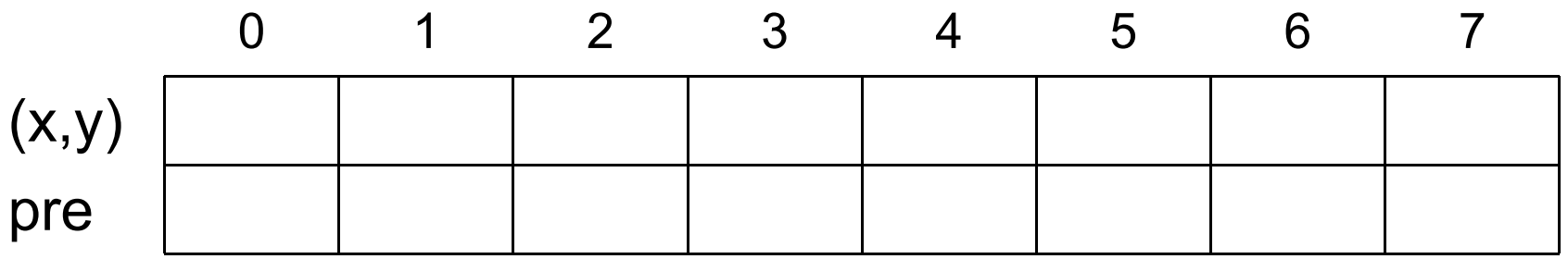
0	0	0	0	0	0	0	0
0	1	1	1	1	0	1	0
0	0	0	0	1	0	1	0
0	1	0	0	0	0	1	0
0	1	0	1	1	0	1	0
0	1	0	0	0	0	1	1
0	1	0	0	1	0	0	0
0	1	1	1	1	1	1	0

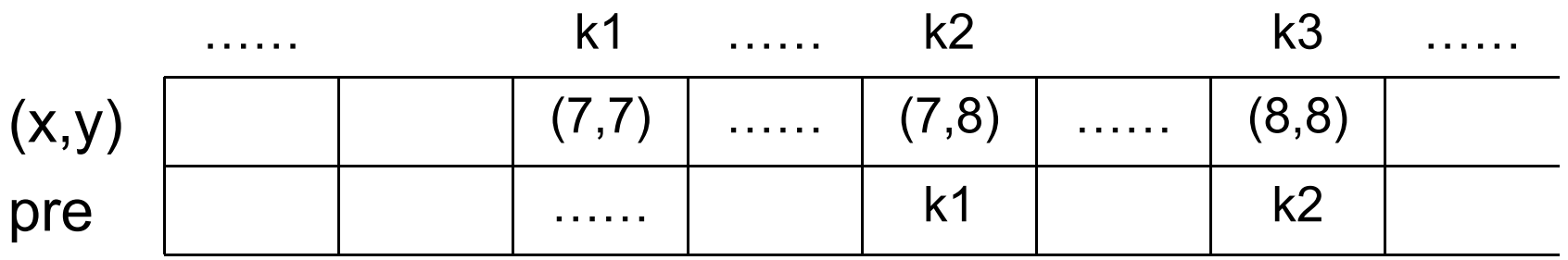
4.3 图的搜索算法

- 算法设计：
  - 每个入队列的方格包含三个属性，x坐标、y坐标、pre前驱方格
  - 将方格（1,1）入队，队首front置为0、队尾rear置为1
  - 将队首方格上下左右四个方向可通行的方格均入队
  - 把已经访问过的方格值改为-1，不另外开辟空间记录其访问的情况
  - 队首front加1，得到新的队首方格。重复此步骤，直到方格（8,8）入队为止，搜索结束。
  - 通过pre就可以倒推出最短线路。

4.3 图的搜索算法


"F:\C运行...
迷宫：（1表示墙）
0 0 0 0 0 0 0 0
0 1 1 1 1 0 1 0
0 0 0 0 1 0 1 0
0 1 0 0 0 0 1 0
0 1 0 1 1 0 1 0
0 1 0 0 0 0 1 1
0 1 0 0 1 0 0 0
0 1 1 1 1 1 1 0
路径示意图：
0 * * * * * * *
0 * * * * * * *
0 0 0 * * * * *
* * 0 * * * * *
* * 0 * * * * *
* * 0 0 0 0 * *
* * * * * 0 0 0
* * * * * * * 0
总步数为：15


0
1
2
3
4
5
6
7
(x,y)
pre


……
k1
……
k2
k3
……
(x,y)
(7,7)
……
(7,8)
……
(8,8)
pre
……
k1
k2

4.3 图的搜索算法

- 算法实现：

```
int
a[9][9]={{0},{0,0,0,0,0,0,0,0,0},{0,0,1,1,1,1,0,1,0},{0,0,0,0,0,1,0,1,0},
{0,0,1,0,0,0,0,1,0},{0,0,1,0,1,1,0,1,0},{0,0,1,0,0,0,0,1,1},{0,0,1,0,0,1,
0,0,0},{0,0,1,1,1,1,1,1,0}};
int fx[5]={0,1,-1,0,0},fy[5]={0,0,0,-1,1};
int front,rear,i,j,k,total=0;
struct{
        int x,y,pre;
}queue[100];  //定义数组当队列
```

4.3 图的搜索算法

```
void out()
{
    a[queue[rear].x][queue[rear].y]=4;  //把（8,8）方块值赋为4
    while(queue[rear].pre!=0)
    {
        rear=queue[rear].pre;   //路径上的方块都赋为4
        a[queue[rear].x][queue[rear].y]=4;
    }
        printf("\n路径示意图：\n");
        for(i=1;i<=8;i++)
        {       for(j=1;j<=8;j++)
                        if(a[i][j]==4)
                        {       printf("0 ");
                                total++;
                        }
                        else
                                printf("* ");
                printf("\n");
        }
}
```

4.3 图的搜索算法

```
int check(int i,int j)
{
    int flag=1;
    if(i<1||i>8||j<1||j>8)    //若坐标移动超出了迷宫范围
        flag=0;
    if(a[i][j]==1||a[i][j]==-1)   //若方块为墙或已搜索过了
                flag=0;
    return(flag);
}

void search()
{
    queue[1].pre=0;
    queue[1].x=1;
    queue[1].y=1;       //（1,1）方格入队列，前驱方格设为0
    front=0;
    rear=1;       //rear对应此方格在队列中的下标
    a[1][1]=-1;      //（1,1）设置为已搜索过的方块
```

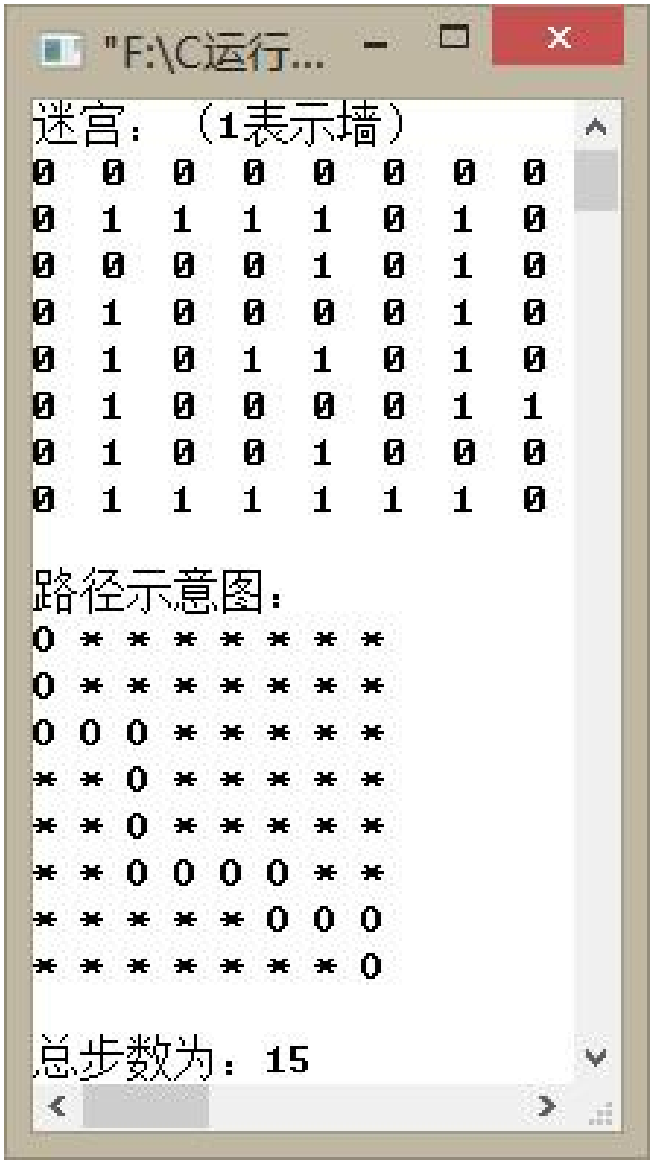
4.3 图的搜索算法

```
    while(front!=rear)     //当队不为空
    {   front=front+1;     //出队
        for(k=1;k<=4;k++) //搜索该出队方块四个方向可通行的方格
        {   i=queue[front].x+fx[k];
            j=queue[front].y+fy[k];
            if(check(i,j)==1)  //如果某一方向可通行
            {                  //把该方向的方格入队列
                rear=rear+1;   //rear对应此方格在队列中的下标
                queue[rear].x=i;
                queue[rear].y=j;
                queue[rear].pre=front;
                a[i][j]=-1;    //设置为已搜索过的方块
                if(queue[rear].x==8&&queue[rear].y==8)
                {   out();
                    return;
                }
            }
        }
    }
}
```

4.3 图的搜索算法

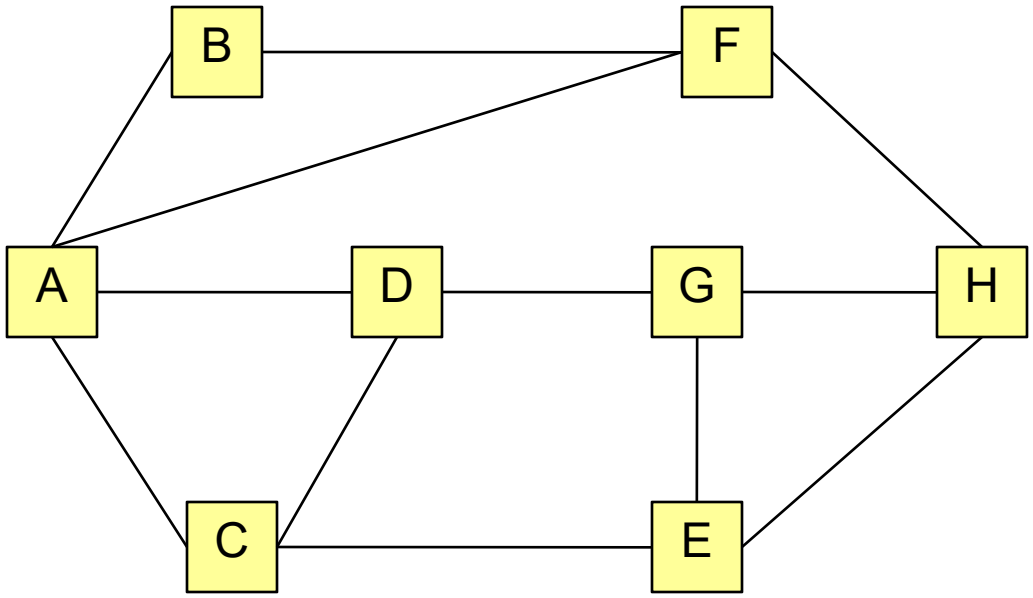
```
int main()
{
        printf("迷宫：（1表示墙）\n");
        for(i=1;i<=8;i++)
                {
                for(j=1;j<=8;j++)
                        printf("%-3d",a[i][j]);
                printf("\n");
                }
    search();
    printf("\n总步数为：%d\n",total);
    return 0;
}
```

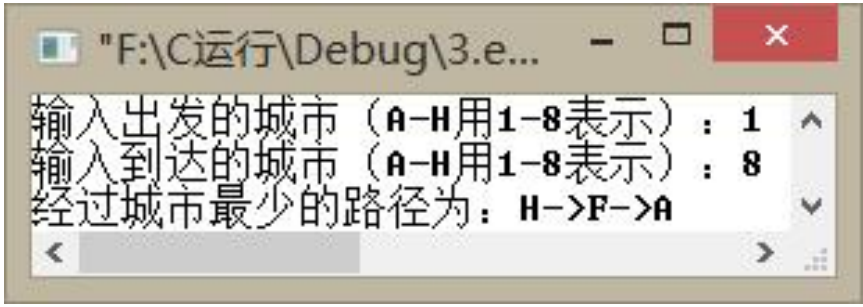
4.3 图的搜索算法

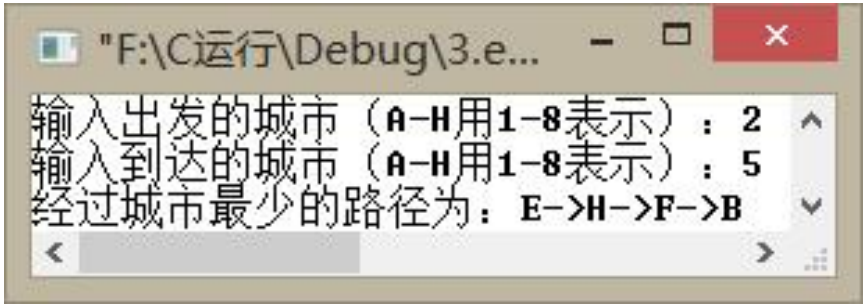

"F:\C运行...
迷宫：（1表示墙）
0 0 0 0 0 0 0 0
0 1 1 1 1 0 1 0
0 0 0 0 1 0 1 0
0 1 0 0 0 0 1 0
0 1 0 1 1 0 1 0
0 1 0 0 0 0 1 1
0 1 0 0 1 0 0 0
0 1 1 1 1 1 1 0
路径示意图：
0 * * * * * * *
0 * * * * * * *
0 0 0 * * * * *
* * 0 * * * * *
* * 0 * * * * *
* * 0 0 0 0 * *
* * * * * 0 0 0
* * * * * * * 0
总步数为：15

4.3 图的搜索算法

- 例：已知若干个城市的地图，求从一个城市到另一个城市的路径，要求路径中经过的城市最少。







4.3 图的搜索算法

- 问题分析：
  - 将起点城市（编号m）入队，队首front置为0、队尾rear置为1
  - 将队首所指的城市所有可直通的城市入队（如果这个城市在队中出现过就不入队）,然后将队首加1，得到新的队首城市。重复以上步骤，直到终点城市（编号n）入队为止。搜索结束。
  - 输出最少城市线路。

4.3 图的搜索算法

- 算法设计：
  - 用矩阵a存储城市间的连接情况
  - 数组queue存储出入队城市的信息。包含两个成员：queue[i].city记录入队的城市号，queue[i].pre记录该入队城市的前趋城市号，通过queue[i].pre就可以倒推出最短线路。
  - 设置数组visited[]记录已搜索过的城市。

4.3 图的搜索算法

```
int a[9][9]={……}; //定义邻接矩阵
struct
{ …… }queue[9];  //定义存储出入队城市信息的结构体数组
int visited[9];  //标识城市是否已搜索过
void out()  //从到达城市开始通过pre倒推输出路径，直到城市编号值-1为止
{ ……}
void search()
{
        ……  //出发城市入queue队列，前驱城市编号设为-1
        while(……)  //队列不空
        {
                ……  //队头城市出队列
                for(……)  //查出队列城市的邻接矩阵信息
                    if (……)  //如果某城市与该出队列城市邻接并且未搜索过
                        {
                                ……  //该城市入队列
                                if(……)  //如果该城市已经是到达城市了
                                        out( );
                        }
        }
}
```

4.3 图的搜索算法

■算法实现：

```
int
a[9][9]={{0}, {0, 0, 1, 1, 1, 0, 1, 0, 0}, {0, 1, 0, 0, 0, 0, 1, 0, 0}, {0, 1, 0, 0, 1, 1, 0, 0, 0},
{0, 1, 0, 1, 0, 0, 0, 1, 0}, {0, 0, 0, 1, 0, 0, 0, 1, 1}, {0, 1, 1, 0, 0, 0, 0, 0, 1}, {0, 0, 0, 0, 1, 1,
0, 0, 1}, {0, 0, 0, 0, 0, 1, 1, 1, 0}};

struct
{
        int city,pre;
}queue[9];

int m,n,front,rear,i,visited[9]={0};
```

4.3 图的搜索算法

```
void out()
{
        printf("经过城市最少的路径为：");
        switch(queue[rear].city)
        {
                case 1:printf("A");break;
                case 2:printf("B");break;
                case 3:printf("C");break;
                case 4:printf("D");break;
                case 5:printf("E");break;
                case 6:printf("F");break;
                case 7:printf("G");break;
                case 8:printf("H");break;
        }
```

4.3 图的搜索算法

```
        while(queue[rear].pre!=-1)
        {
                rear=queue[rear].pre;
                switch(queue[rear].city)
                {
                        case 1:printf("->A");break;
                        case 2:printf("->B");break;
                        case 3:printf("->C");break;
                        case 4:printf("->D");break;
                        case 5:printf("->E");break;
                        case 6:printf("->F");break;
                        case 7:printf("->G");break;
                        case 8:printf("->H");break;
                }
        }
        printf("\n");
}
```

4.3 图的搜索算法

```
void search()
{       front=0;
        rear=1;
        queue[1].city=m;
        queue[1].pre=-1;
        visited[m]=1;
        while(front!=rear)
        {       front=front+1;
                for(i=1;i<=8;i++)
                        if ((a[queue[front].city][i]==1)&&(visited[i]==0))
                        {
                                rear=rear+1;
                                queue[rear].city=i;
                                queue[rear].pre=front;
                                visited[i]=1;
                                if(queue[rear].city==n)
                                {       out( );
                                        return;
                                }
                        }
        }
}
```

4.3 图的搜索算法

```
int main()
{
        printf("输入出发的城市（A-H用1-8表示）：");
        scanf("%d",&m);
        printf("输入到达的城市（A-H用1-8表示）：");
        scanf("%d",&n);
        search();
        return 0;
}
```