



# Data Structures

Ch6



# Graphs

2022 年 11 月 2 日

学而不厌 诲人不倦

- ➡ 6.1 引言
- ➡ 6.2 图的逻辑结构
- ➡ **6.3 图的存储结构及实现**
- ➡ 6.4 最小生成树
- ➡ 6.5 最短路径
- ➡ 6.6 有向无环图及其应用
- ➡ 6.7 扩展与提高
- ➡ 6.8 应用实例

## 6.3 图的存储结构及实现

### 6-3-1 图的邻接矩阵存储结构

## 6.3 图的存储结构及实现

### 6-3-1 图的邻接矩阵存储结构

#### 1. 图的存储结构

 图是否可以采用顺序存储结构？

在图中，任何两个顶点之间都**可能存在关系**（边）



无法通过存储位置表示这种**任意的逻辑关系**



图无法采用顺序存储结构

 如何存储图呢？

图是由顶点和边组成



**分别**考虑

{ 如何存储顶点  
如何存储边



#### 1. 图的存储结构

#### 邻接矩阵

 邻接矩阵也称**数组**表示法，其基本思想是：

- 一维数组：存储图中顶点的信息
- 二维数组（邻接矩阵）：存储图中各顶点之间的邻接关系

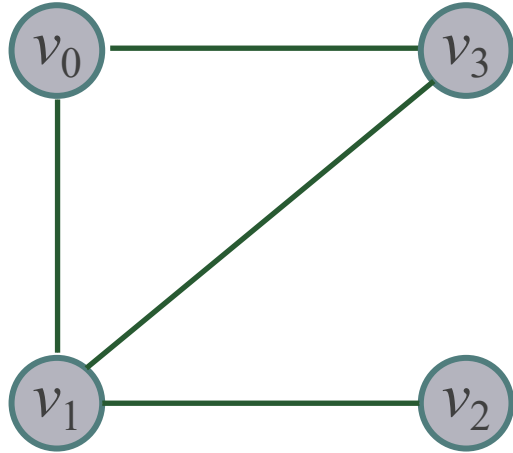
设  $G=(V, E)$  有  $n$  个顶点，则邻接矩阵是一个  $n \times n$  的方阵，定义为：

$$\text{edge}[i][j] = \begin{cases} 1 & \text{若 } (v_i, v_j) \in E \text{ (或 } \langle v_i, v_j \rangle \in E) \\ 0 & \text{其它} \end{cases}$$

## 6.3 图的存储结构及实现

### 6-3-1 图的邻接矩阵存储结构

#### 2. 图的存储示意图



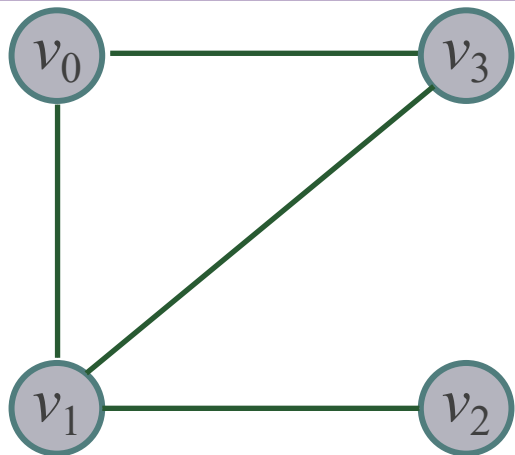
$$\text{vertex} = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{matrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{matrix} & \begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{matrix}$$

- 🕒 无向图的邻接矩阵有什么特点？  $\Rightarrow$  对称矩阵
- 🕒 如何求顶点  $v$  的度？  $\Rightarrow$  第  $v$  行（或第  $v$  列）非零元素的个数
- 🕒 如何判断顶点  $i$  和  $j$  之间是否存在边？  $\Rightarrow$  if (edge[i][j] == 1)

## 6.3 图的存储结构及实现

### 6-3-1图的邻接矩阵存储结构

#### 2. 图的存储示意图



vertex =

	0	1	2	3
	$v_0$	$v_1$	$v_2$	$v_3$

edge =

	$v_0$	$v_1$	$v_2$	$v_3$
$v_0$	0	1	0	1
$v_1$	1	0	1	1
$v_2$	0	1	0	0
$v_3$	1	1	0	0



如何求顶点  $i$  的所有邻接点？



扫描第  $i$  行

```
for (j = 0; j < n; j++)
```

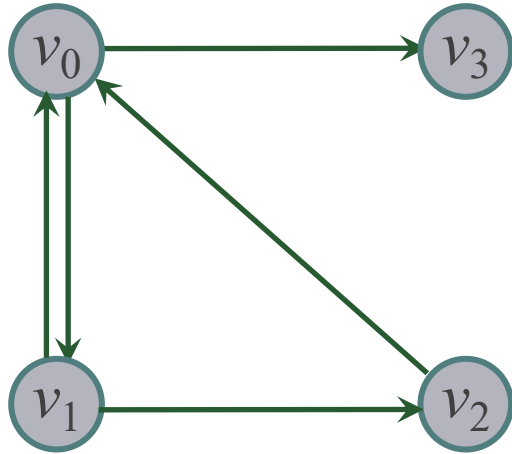
```
    if (edge[i][j] == 1)
```

顶点  $j$  是顶点  $i$  的邻接点

## 6.3 图的存储结构及实现

### 6-3-1图的邻接矩阵存储结构

#### 2. 图的存储示意图



$$\text{vertex} = \begin{matrix} & \begin{matrix} 0 & 1 & 2 & 3 \end{matrix} \\ \begin{matrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{matrix} & \begin{bmatrix} v_0 & v_1 & v_2 & v_3 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

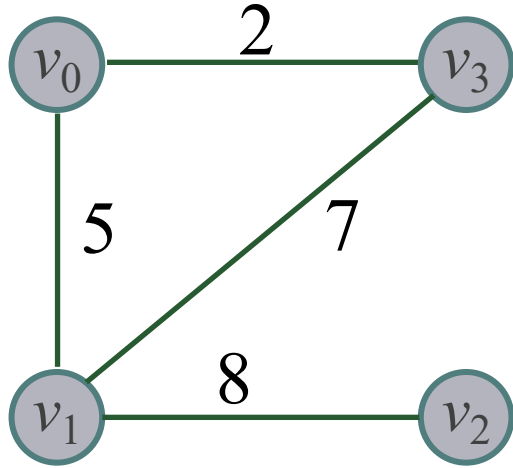
- 🕒 有向图的邻接矩阵一定不对称吗？  $\Rightarrow$  顶点间存在方向相反的弧
- 🕒 如何求顶点  $v$  的出度？  $\Rightarrow$  第  $v$  行非零元素的个数
- 🕒 如何求顶点  $v$  的入度？  $\Rightarrow$  第  $v$  列非零元素的个数



## 6.3 图的存储结构及实现

### 6-3-1图的邻接矩阵存储结构

#### 2. 图的存储示意图



vertex = 

0	1	2	3
$v_0$	$v_1$	$v_2$	$v_3$

edge = 

	$v_0$	$v_1$	$v_2$	$v_3$
$v_0$	0	5	$\infty$	2
$v_1$	5	0	8	7
$v_2$	$\infty$	8	0	$\infty$
$v_3$	2	7	$\infty$	0

网图的邻接矩阵可定义为：

$$\text{arc}[i][j] = \begin{cases} w_{ij} & \text{若 } (v_i, v_j) \in E \text{ (或 } \langle v_i, v_j \rangle \in E) \\ 0 & \text{若 } i = j \\ \infty & \text{其他} \end{cases}$$



## 6.3 图的存储结构及实现

### 6-3-1 图的邻接矩阵存储结构

#### 3. 图的存储结构定义

 图的抽象数据类型定义？

ADT Graph

DataModel

...

Operation

**CreatGraph** : 图的建立

**DestroyGraph** : 图的销毁

**DFTraverse** : 深度优先遍历图

**BFTraverse** : 广度优先遍历图

endADT



```
const int MaxSize = 10;
template <typename DataType>
class MGraph
{
public:
    MGraph(DataType a[ ], int n, int e);
    ~MGraph( );
    void DFTraverse(int v);
    void BFTraverse(int v);
private:
    DataType vertex[MaxSize];
    int edge[MaxSize][MaxSize];
    int vertexNum, edgeNum;
};
```

## 6.3 图的存储结构及实现

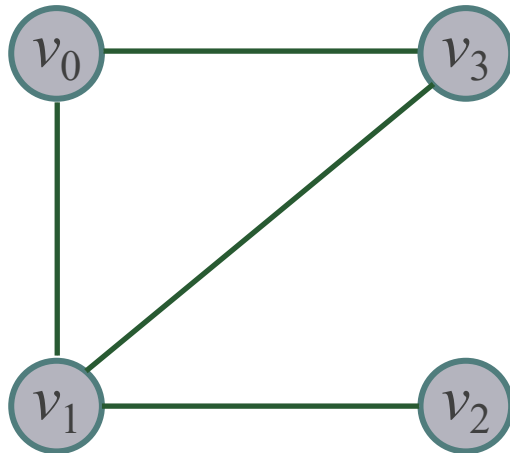
### 6-3-1 图的邻接矩阵存储结构

#### 4. 图的建立

输入

$a =$ 

$v_0$	$v_1$	$v_2$	$v_3$
-------	-------	-------	-------



结果

vertex = 

$v_0$	$v_1$	$v_2$	$v_3$
-------	-------	-------	-------

edge = 

	$v_0$	$v_1$	$v_2$	$v_3$
$v_0$	0	1	0	1
$v_1$	1	0	1	1
$v_2$	0	1	0	0
$v_3$	1	1	0	0

vertexNum    edgeNum

4	4
---	---



## 6.3 图的存储结构及实现

### 6-3-1 图的邻接矩阵存储结构

#### 4. 图的建立

算法：CreatGraph(a[n], n, e)

输入：顶点的数据a[n]，顶点个数n，边的个数e

输出：图的邻接矩阵

1. 存储图的顶点个数和边的个数；
2. 将顶点信息存储在一维数组vertex中；
3. 初始化邻接矩阵edge；
4. 依次输入每条边并存储在邻接矩阵edge中：
  - 4.1 输入边依附的两个顶点的编号i和j；
  - 4.2 将edge[i][j]和edge[j][i]的值置为1；

$$\text{vertex} = \begin{bmatrix} v_0 & v_1 & v_2 & v_3 \end{bmatrix}$$
$$\text{edge} = \begin{matrix} & \begin{matrix} v_0 & v_1 & v_2 & v_3 \end{matrix} \\ \begin{matrix} v_0 \\ v_1 \\ v_2 \\ v_3 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{bmatrix} \end{matrix}$$

vertexNum    edgeNum

4	4
---	---



## 6.3 图的存储结构及实现

### 6-3-1 图的邻接矩阵存储结构

#### 4. 图的建立

```
template <typename DataType>
MGraph<DataType> :: MGraph(DataType a[ ], int n, int e)
{
    int i, j, k;
    vertexNum = n; edgeNum = e;
    for (i = 0; i < vertexNum; i++)           //存储顶点
        vertex[i] = a[i];
    for (i = 0; i < vertexNum; i++)           //初始化邻接矩阵
        for (j = 0; j < vertexNum; j++)
            edge[i][j] = 0;
    for (k = 0; k < edgeNum; k++)             //依次输入每一条边
    {
        cin >> i >> j;                       //输入边依附的两个顶点的编号
        edge[i][j] = 1; edge[j][i] = 1;       //置有边标志
    }
}
```

## 6.3 图的存储结构及实现

### 6-3-1图的邻接矩阵存储结构



#### 5. 图的深度优先遍历

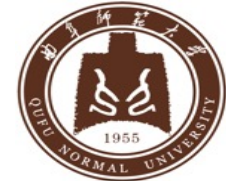
算法：DFTraverse

输入：顶点的编号  $v$

输出：图的深度优先遍历序列

1. 访问顶点  $v$ ; 修改标志  $visited[v] = 1$ ;
2.  $j$  = 顶点  $v$  的第一个邻接点 ;
3. while ( $j$  存在)
  - 3.1 if ( $j$  未被访问) 从顶点  $j$  出发递归执行该算法;
  - 3.2  $j$  = 顶点  $v$  的下一个邻接点;

	0	1	2	3
	$v_0$	$v_1$	$v_2$	$v_3$
$v_0$	0	1	0	1
$v_1$	1	0	1	1
$v_2$	0	1	0	0
$v_3$	1	1	0	0



## 5. 图的深度优先遍历

```
template <typename DataType>
void MGraph<DataType> :: DFTraverse(int v)
{
    cout << vertex[v]; visited[v] = 1;
    for (int j = 0; j < vertexNum; j++)
        if (edge[v][j] == 1 && visited[j] == 0)
            DFTraverse( j );
}
```

	0	1	2	3
	$v_0$	$v_1$	$v_2$	$v_3$
$v_0$	0	1	0	1
$v_1$	1	0	1	1
$v_2$	0	1	0	0
$v_3$	1	1	0	0

## 6.3 图的存储结构及实现

### 6-3-1图的邻接矩阵存储结构

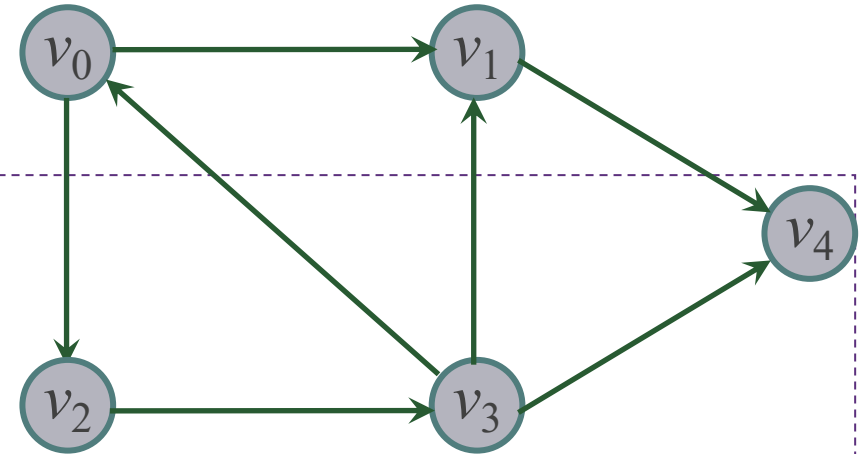
## 6. 图的广度优先遍历

算法：BFTraverse

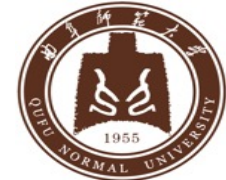
输入：顶点的编号  $v$

输出：图的广度优先遍历序列

1. 队列  $Q$  初始化;
2. 访问顶点  $v$ ; 修改标志  $\text{visited}[v] = 1$ ; 顶点  $v$  入队列  $Q$ ;
3. while (队列  $Q$  非空)
  - 3.1  $i$  = 队列  $Q$  的队头元素出队;
  - 3.2  $j$  = 顶点  $v$  的第一个邻接点;
  - 3.3 while ( $j$  存在)
    - 3.3.1 如果  $j$  未被访问, 则  
访问顶点  $j$ ; 修改标志  $\text{visited}[j] = 1$ ; 顶点  $j$  入队列  $Q$ ;
    - 3.3.2  $j$  = 顶点  $i$  的下一个邻接点;







## 6. 图的广度优先遍历

```
template <typename DataType>
void MGraph<DataType> :: BFTraverse(int v)
{
    int w, j, Q[MaxSize];                                //采用顺序队列
    int front = -1, rear = -1;                             //初始化队列
    cout << vertex[v]; visited[v] = 1; Q[++rear] = v;      //被访问顶点入队
    while (front != rear)                                   //当队列非空时
    {
        w = Q[++front];                                     //将队头元素出队并送到v中
        for (j = 0; j < vertexNum; j++)
            if (edge[w][j] == 1 && visited[j] == 0 ) {
                cout << vertex[j]; visited[j] = 1; Q[++rear] = j;
            }
    }
}
```

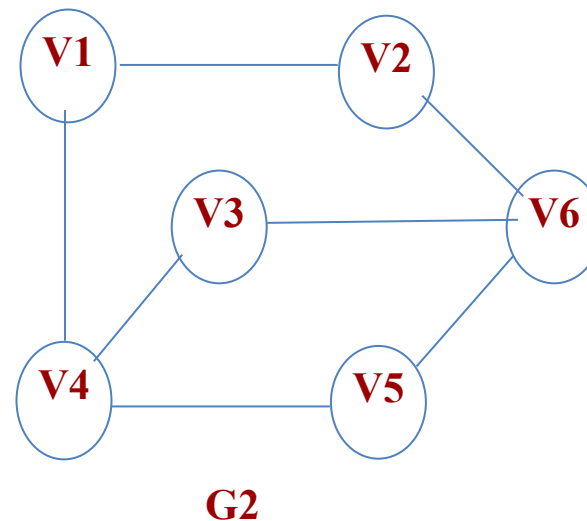
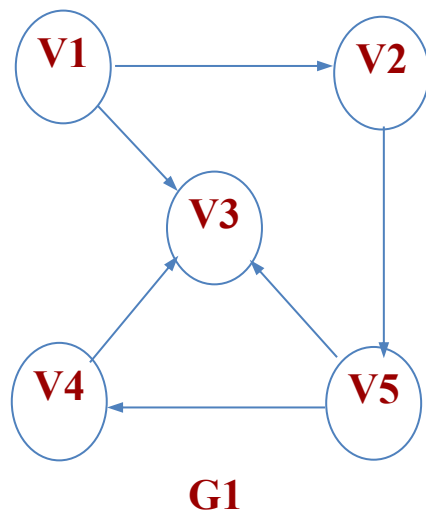
## 6.3 图的存储结构及实现

### 6-3-1 图的邻接矩阵存储结构



#### 例题

1. 有向图G1和无向图G2如下图所示，请分别画出它们的邻接矩阵和邻接表。



## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

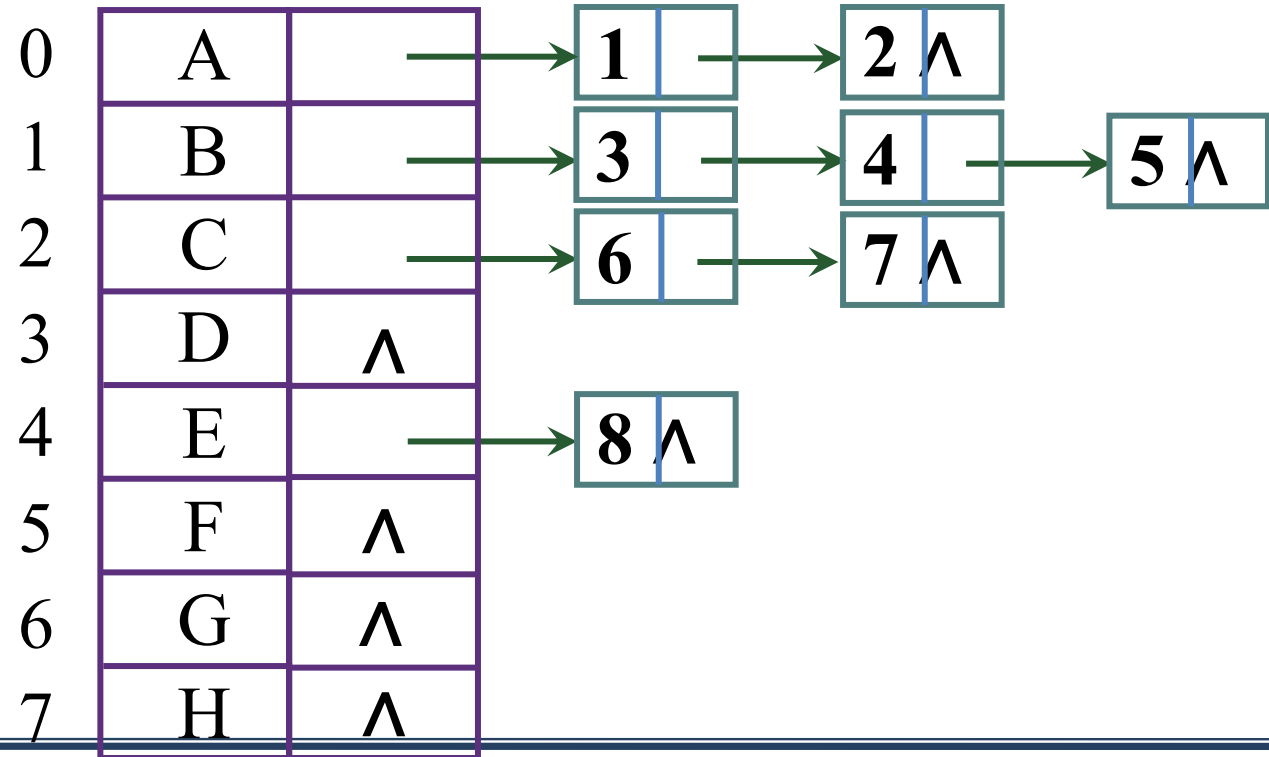
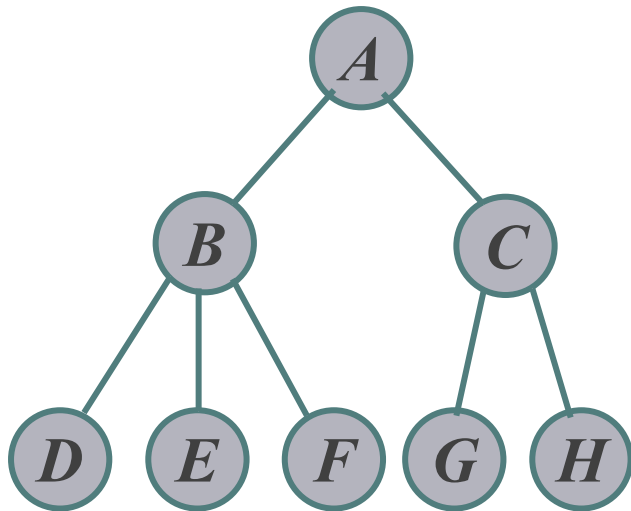
2022 年 11 月 11 日

## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 1. 图的邻接表存储

- 🕒 邻接矩阵存储结构的空间复杂度是多少？  $\Rightarrow O(n^2)$
- 🕒 如果采用邻接矩阵存储**稀疏图**，会出现什么情况？  $\Rightarrow$  稀疏矩阵
- 🕒 树的孩子表示法？



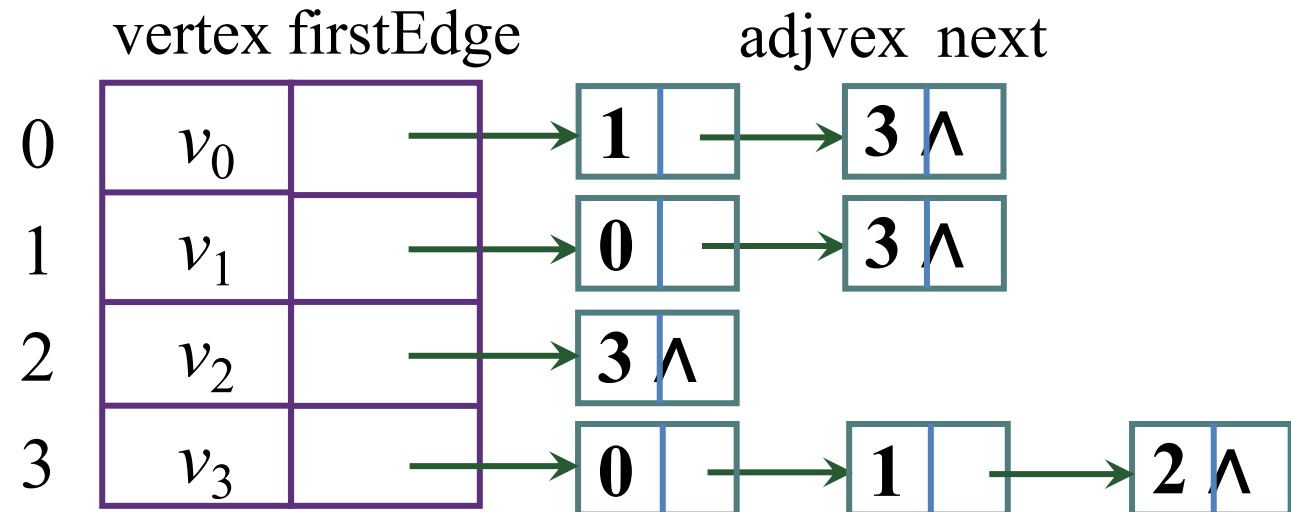
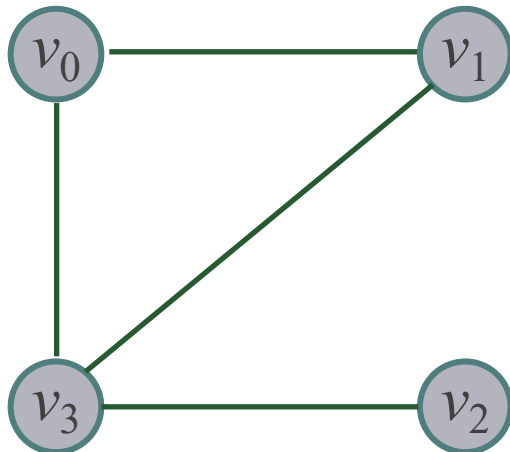
## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 1. 图的邻接表存储

 邻接表存储的基本思想是：

- 边表（邻接表）：顶点  $v$  的所有邻接点链成的单链表
- 顶点表：所有边表的头指针和存储顶点信息的一维数组



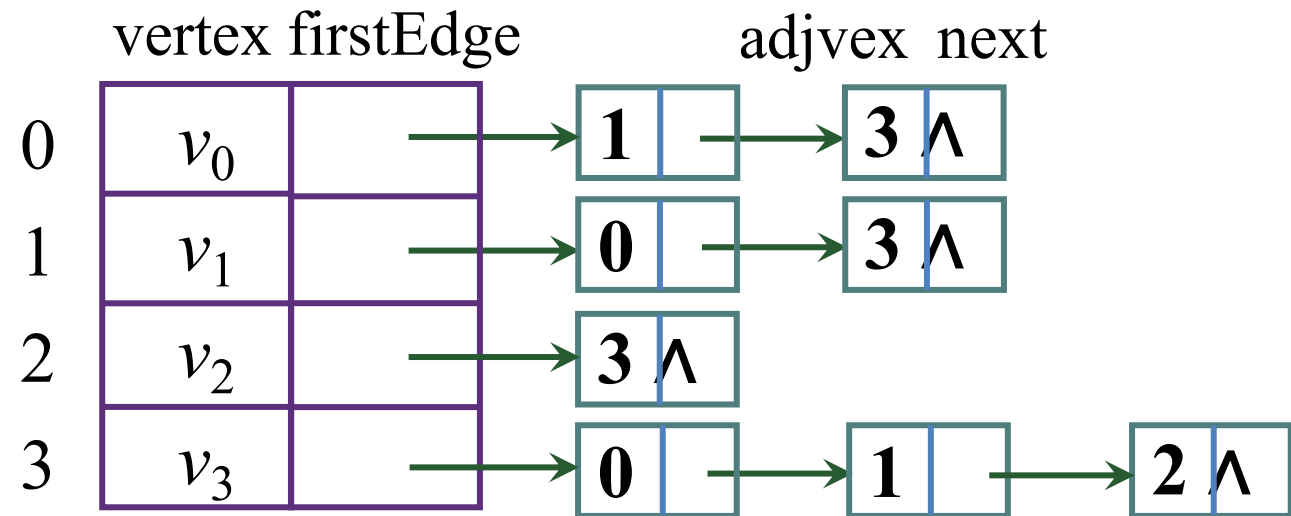
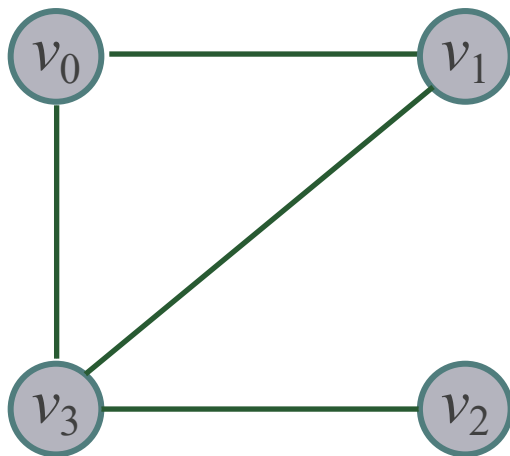
## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 2. 图的邻接表存储结构定义

```
struct EdgeNode
{
    int adjvex;
    EdgeNode *next;
};
```

```
template <typename DataType>
struct VertexNode
{
    DataType vertex;
    EdgeNode *firstEdge;
};
```



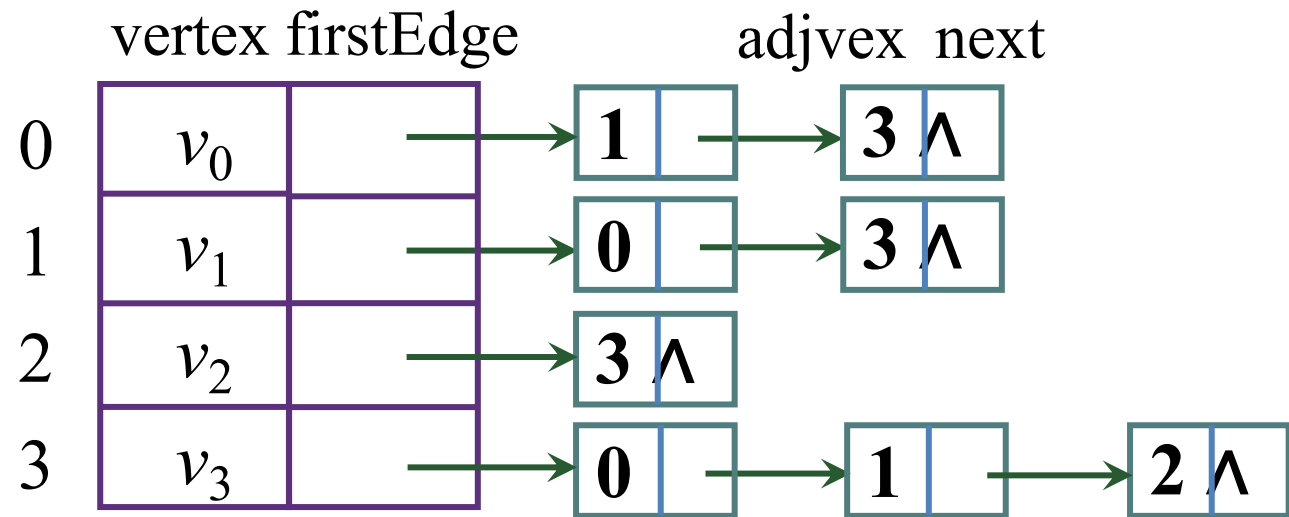
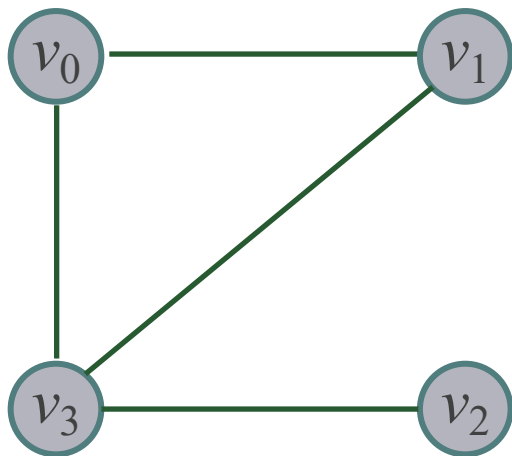
## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 3. 图的邻接表基本操作

🕒 边表中的结点表示什么？  $\Rightarrow$  对应图中的一条边

🕒 设图有 $n$ 个顶点 $e$ 条边，邻接表的空间复杂度是多少？  $\Rightarrow O(n+e)$



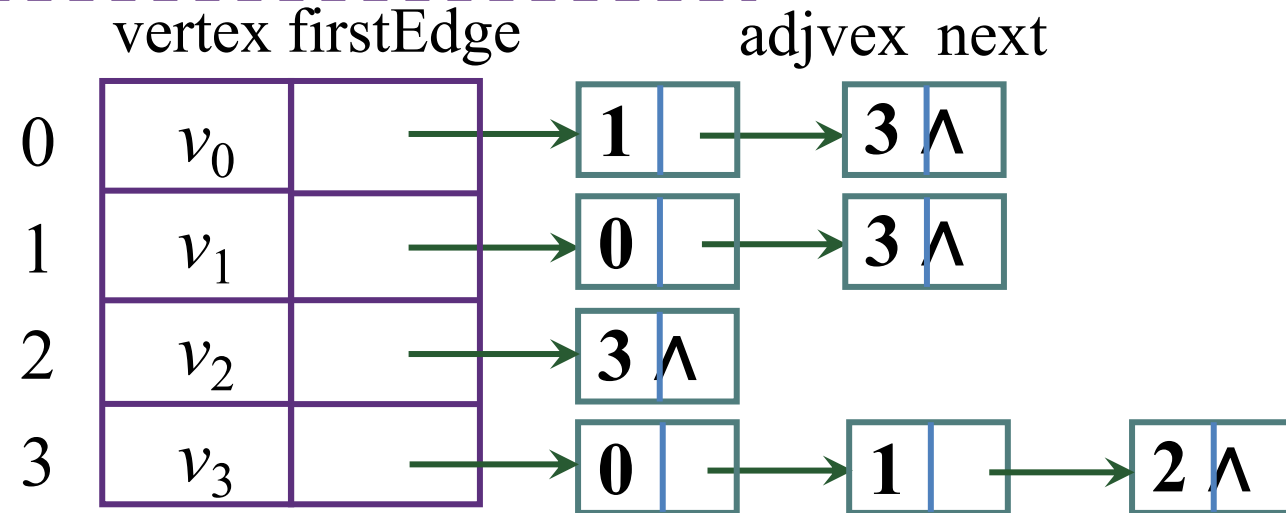
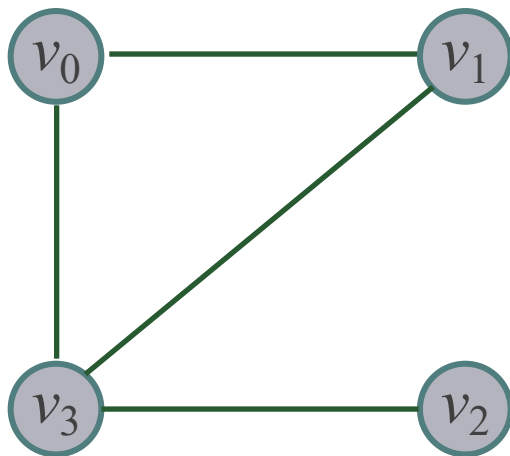
## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 3. 图的邻接表基本操作

 如何求顶点  $v$  的度？  $\Rightarrow$  顶点  $v$  的边表中结点的个数

```
p = adjlist[v].firstEdge; count = 0;
while (p != nullptr)
{
    count++; p = p->next;
}
```





## 6.3 图的存储结构及实现

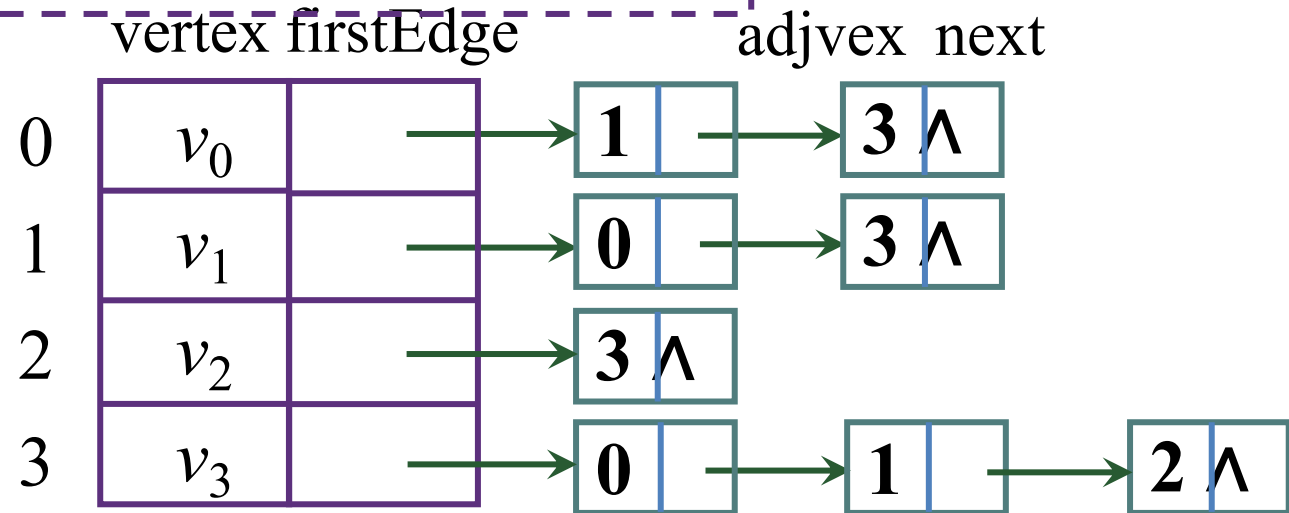
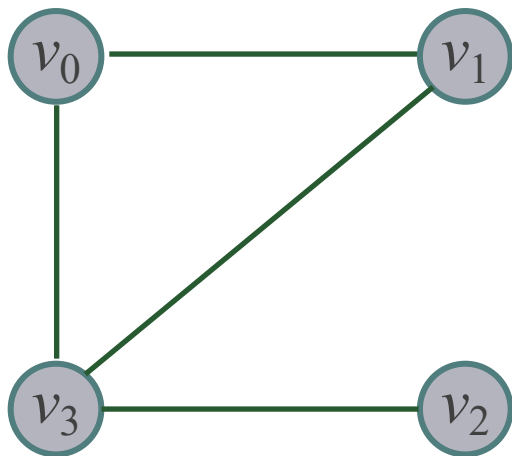
### 6-3-2 图的邻接表存储结构

#### 3. 图的邻接表基本操作

🕒 如何求顶点  $v$  的所有邻接点?  $\Rightarrow$  顶点  $i$  的边表中的所有结点

```
p = adjlist[v].firstEdge;
while (p != nullptr)
{
    j = p->adjvex;
    p = p->next;
}
```

//j是v的邻接点



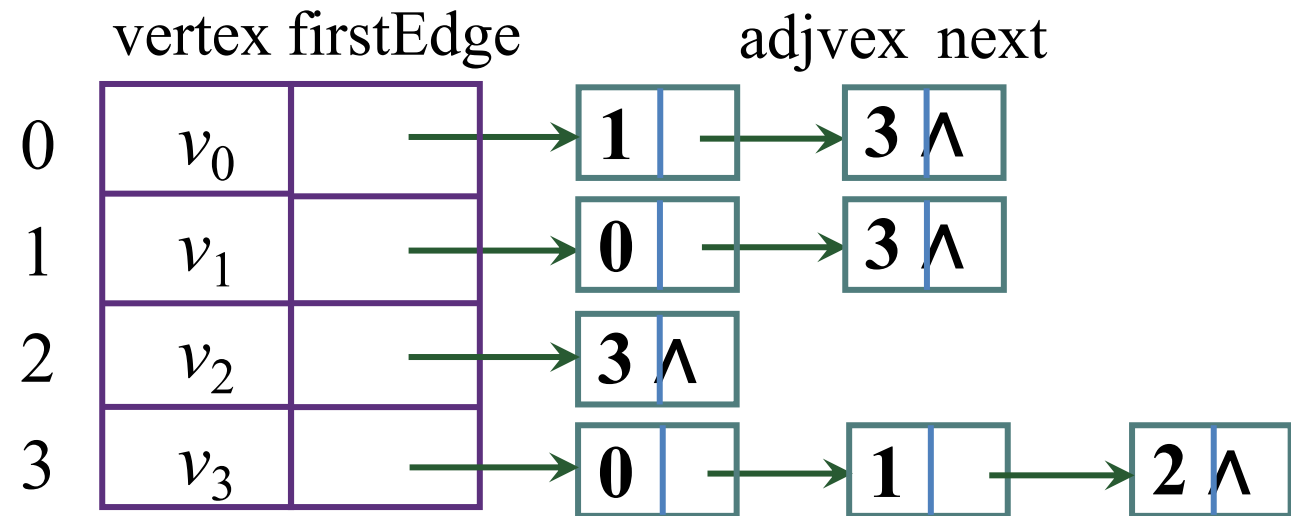
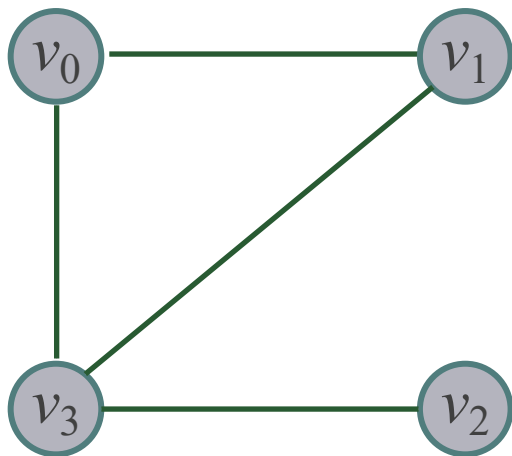
## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 3. 图的邻接表基本操作

 如何判断顶点  $i$  和顶点  $j$  之间是否存在边?

⇒ 测试顶点  $i$  的边表中是否存在数据域为  $j$  的结点



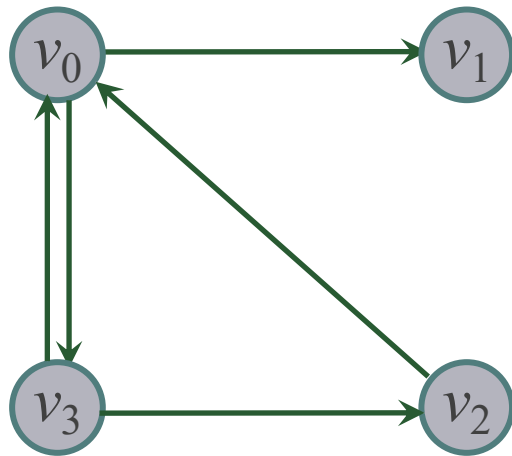
## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 4. 存储有向图

🕒 如何求顶点  $v$  的出度？  $\Rightarrow$  顶点  $v$  的出边表中结点的个数

🕒 如何求顶点  $v$  的入度？  $\Rightarrow$  所有出边表中数据域为  $v$  的结点个数



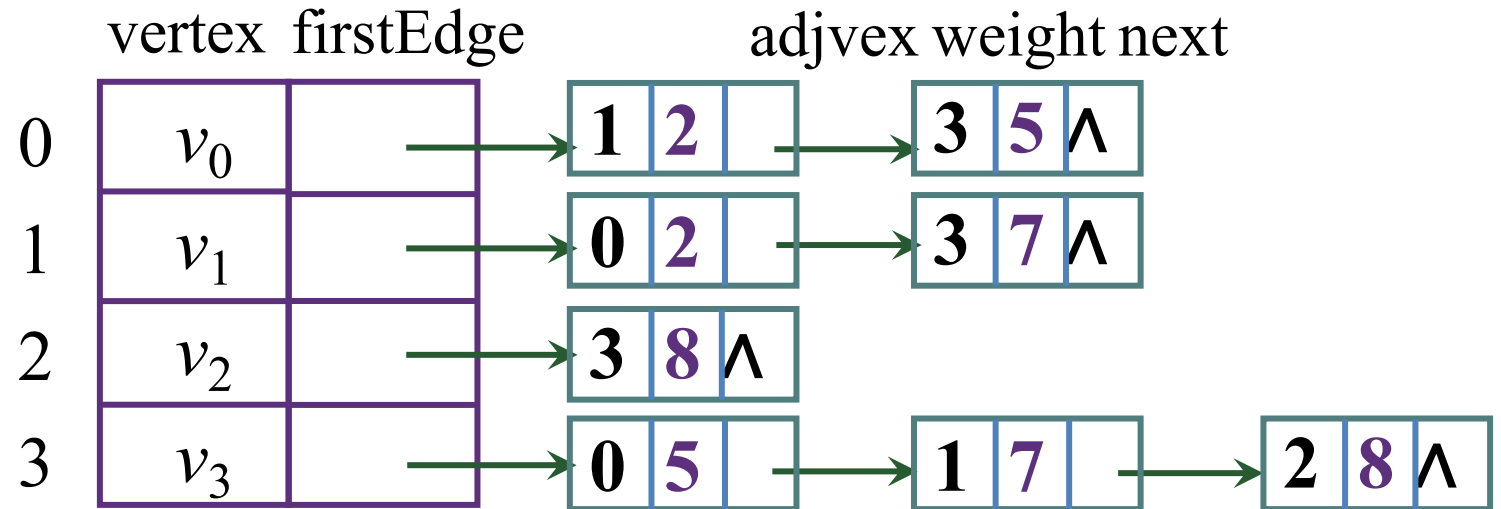
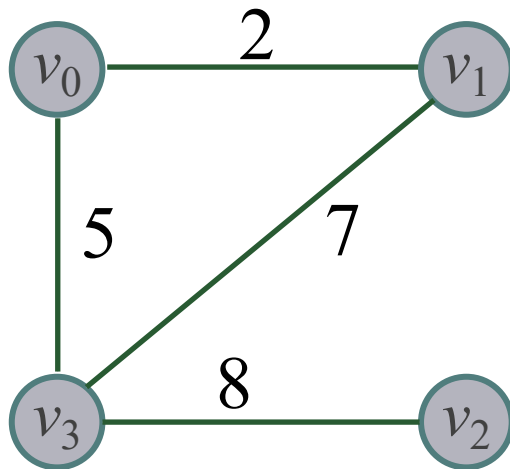
	vertex	firstEdge	adjvex	next
0	$v_0$		1	3 $\wedge$
1	$v_1$	$\wedge$		
2	$v_2$		0 $\wedge$	
3	$v_3$		0	2 $\wedge$

## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 5. 存储带权图

🕒 邻接表如何存储带权图？  $\Rightarrow$  权值存储在边表中



## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 6. 邻接表存储结构定义

 图的抽象数据类型定义？

ADT Graph

DataModel

...

Operation

CreatGraph : 图的建立

DestroyGraph : 图的销毁

DFTraverse : 深度优先遍历图

BFTraverse : 广度优先遍历图

endADT



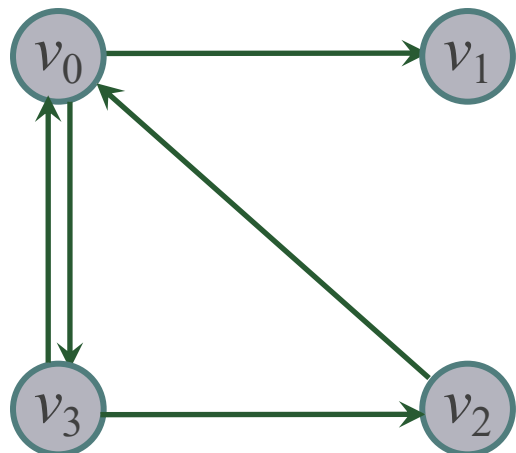
```
const int MaxSize = 10;
template <typename DataType>
class ALGraph
{
public:
    ALGraph(DataType a[ ], int n, int e);
    ~ALGraph( );
    void DFTraverse(int v);
    void BFTraverse(int v);
private:
    VertexNode<DataType> adjlist[MaxSize];
    int vertexNum, edgeNum;
};
```



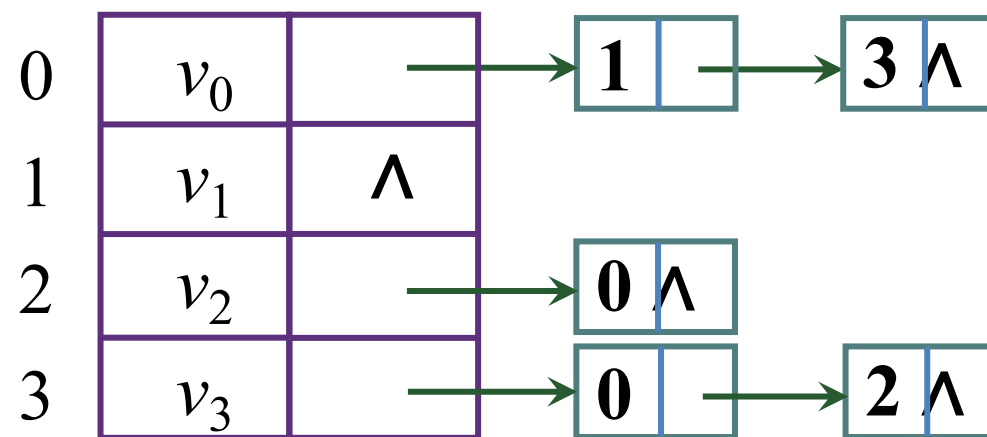
## 7. 图的建立

输入

$a = [v_0, v_1, v_2, v_3]$



结果





## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

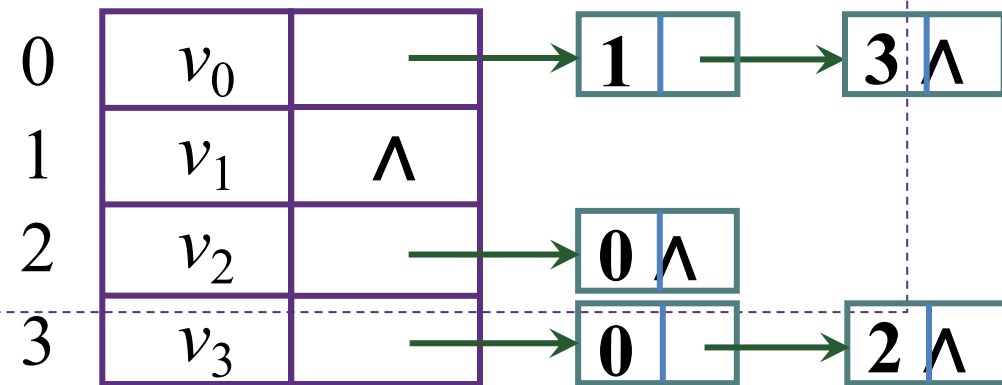
#### 7. 图的建立

算法：CreatGraph(a[n], n, e)

输入：顶点的数据信息a[n]，顶点个数 n，边的个数 e

输出：图的邻接表

1. 存储图的顶点个数和边的个数；
2. 将顶点信息存储在顶点表中，将该顶点边表的头指针初始化为NULL；
3. 依次输入边的信息并存储在边表中：
  - 3.1 输入边所依附的两个顶点的编号 i 和 j；
  - 3.2 生成边表结点 s，其邻接点的编号为 j；
  - 3.3 将结点 s 插入到第 i 个边表的表头；





## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 7. 图的建立

```
template <typename DataType>
ALGraph<DataType> :: ALGraph(DataType a[ ], int n, int e)
{
    int i, j, k;  EdgeNode *s = nullptr;
    vertexNum = n; edgeNum = e;
    for (i = 0; i < vertexNum; i++)    //输入顶点信息，初始化顶点表
    {
        adjlist[i].vertex = a[i]; adjlist[i].firstEdge = nullptr;
    }
}
```

$v_0$	$\wedge$
$v_1$	$\wedge$
$v_2$	$\wedge$
$v_3$	$\wedge$

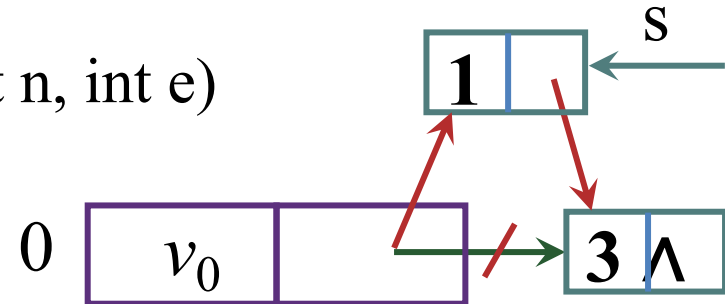


## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 7. 图的建立

```
template <typename DataType>
ALGraph<DataType> :: ALGraph(DataType a[ ], int n, int e)
{
    int i, j, k;   EdgeNode *s = nullptr;
    vertexNum = n; edgeNum = e;
    for (i = 0; i < vertexNum; i++)    //输入顶点信息，初始化顶点表
    {
        adjlist[i].vertex = a[i]; adjlist[i].firstEdge = nullptr;
    }
    for (k = 0; k < edgeNum; k++)    //依次输入每一条边
    {
        cin >> i >> j;                //输入边所依附的两个顶点的编号
        s = new EdgeNode; s->adjvex = j; //生成一个边表结点s
        s->next = adjlist[i].firstEdge; //将结点s插入到第i个边表的表头
        adjlist[i].firstEdge = s;
    }
}
```

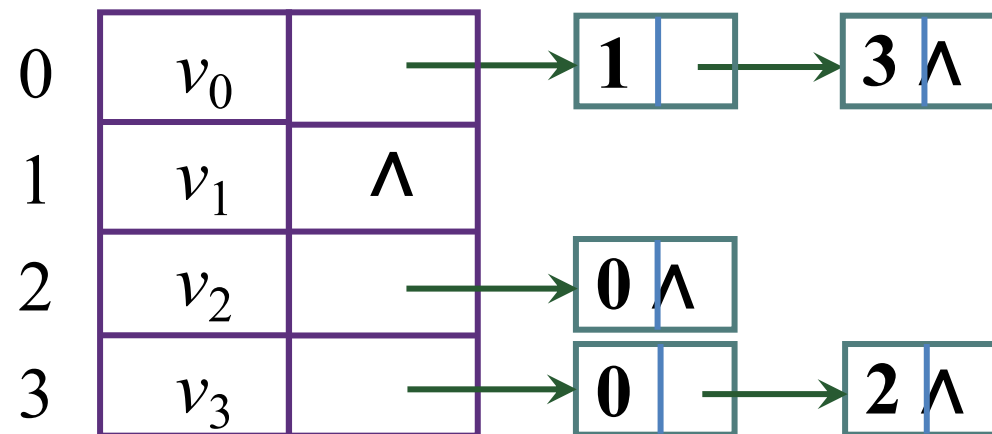




## 8. 图的销毁

在邻接表存储中，须释放所有在程序运行过程中申请的边表结点

```
template <typename DataType>
ALGraph<DataType>::~~ALGraph( )
{
    EdgeNode *p = nullptr, *q = nullptr;
    for (int i = 0; i < vertexNum; i++)
    {
        p = q = adjlist[i].firstEdge;
        while (p != nullptr)
        {
            p = p->next;
            delete q; q = p;
        }
    }
}
```



## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

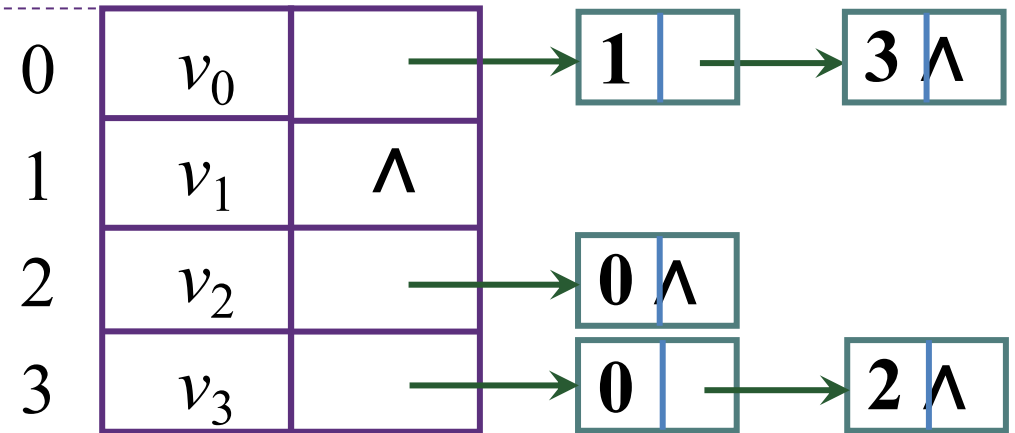
#### 9. 图的深度优先遍历

算法：DFTraverse

输入：顶点的编号  $v$

输出：无

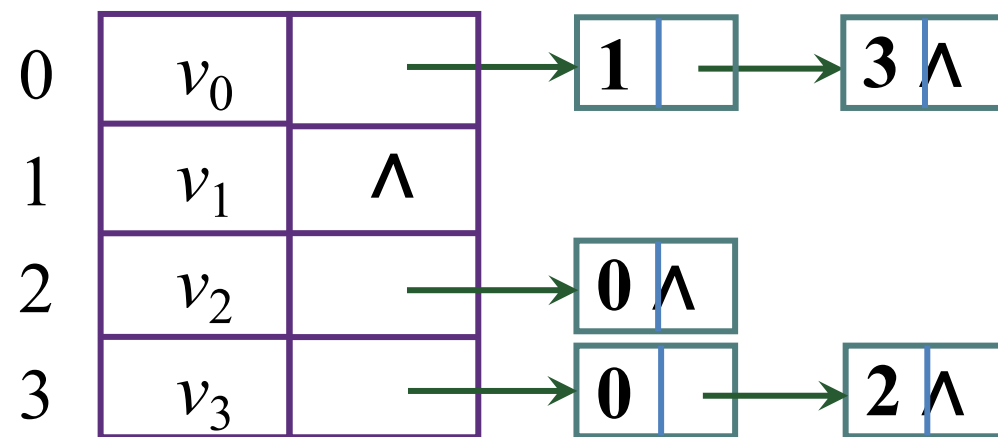
1. 访问顶点  $v$ ; 修改标志  $visited[v] = 1$ ;
2.  $j$  = 顶点  $v$  的第一个邻接点;
3. while ( $j$  存在)
  - 3.1 if ( $j$  未被访问) 从顶点  $j$  出发递归执行该算法;
  - 3.2  $j$  = 顶点  $v$  的下一个邻接点;

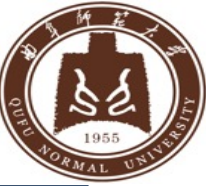




## 9. 图的深度优先遍历

```
template <typename DataType>
void ALGraph<DataType> :: DFTraverse(int v)
{
    int j; EdgeNode *p = nullptr;
    cout << adjlist[v].vertex; visited[v] = 1;
    p = adjlist[v].firstEdge;
    while (p != nullptr)
    {
        j = p->adjvex;
        if (visited[j] == 0) DFTraverse(j);
        p = p->next;
    }
}
```





## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 10. 图的广度优先遍历

算法：BFTraverse

输入：顶点的编号  $v$

输出：无

1. 队列  $Q$  初始化;
2. 访问顶点  $v$  ; 修改标志  $visited[v] = 1$ ; 顶点  $v$  入队列  $Q$ ;
3. while (队列  $Q$  非空)
  - 3.1  $i$  = 队列  $Q$  的队头元素出队;
  - 3.2  $j$  = 顶点  $v$  的第一个邻接点;
  - 3.3 while ( $j$  存在)
    - 3.3.1 如果  $j$  未被访问, 则  
访问顶点  $j$  ; 修改标志  $visited[j] = 1$ ; 顶点  $j$  入队列  $Q$  ;
    - 3.3.2  $j$  = 顶点  $i$  的下一个邻接点 ;

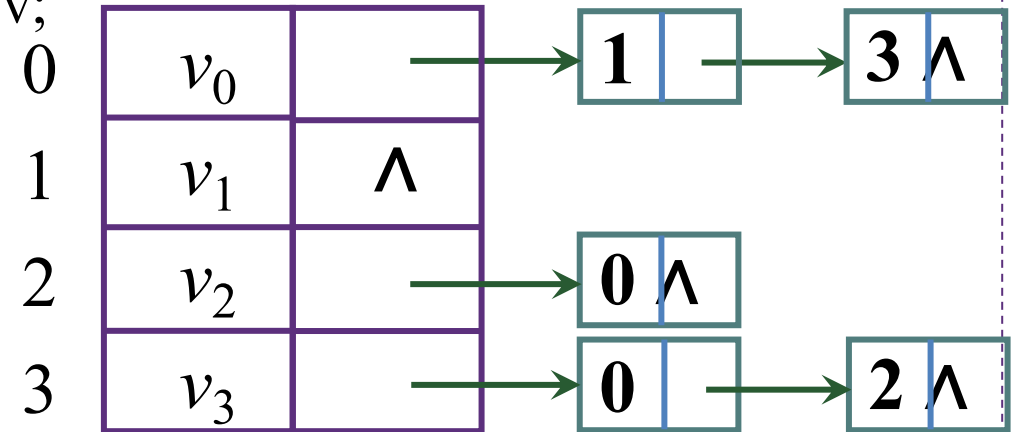
## 6.3 图的存储结构及实现

### 6-3-2 图的邻接表存储结构

#### 10. 图的广度优先遍历

```

template <typename DataType>
void ALGraph<DataType> :: BFTraverse(int v)
{
    int w, j, Q[MaxSize]; int front = -1, rear = -1;
    EdgeNode *p = nullptr;
    cout << adjlist[v].vertex; visited[v] = 1; Q[++rear] = v;
    while (front != rear)
    {
        w = Q[++front];
        p = adjlist[w].firstEdge;
        while (p != nullptr)
        {
            j = p->adjvex;
            if (visited[j] == 0) {
                cout << adjlist[j].vertex; visited[j] = 1; Q[++rear] = j;
            }
            p = p->next;
        }
    }
}
    
```

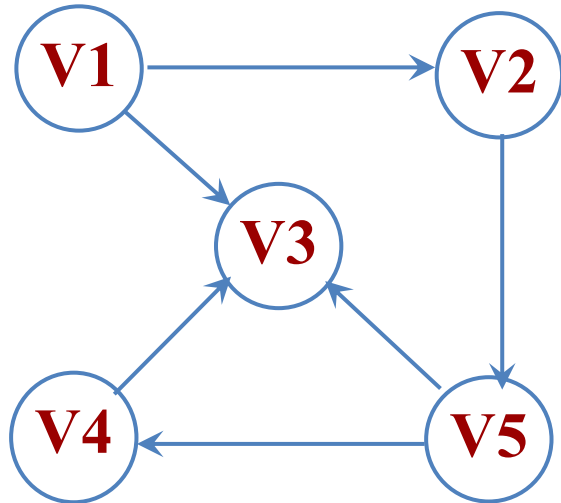


## 小结

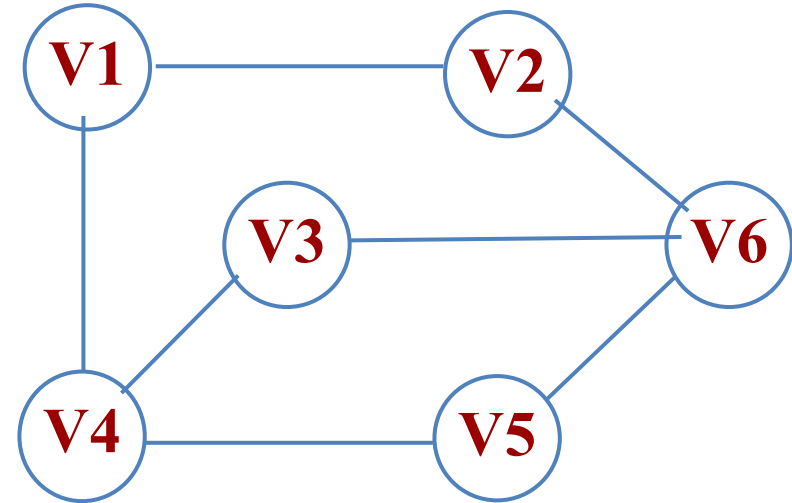
1. 熟练掌握图的邻接矩阵存储及实现方法
2. 掌握图的邻接表存储及实现方法
3. 理解图的邻接矩阵与邻接表实现方法的区别
4. 掌握图的建立、深度优先遍历、广度优先遍历实现方法

# 作业

2. 有向图G1和无向图G2如下图所示，请分别画出它们的邻接矩阵和邻接表。



**G1**

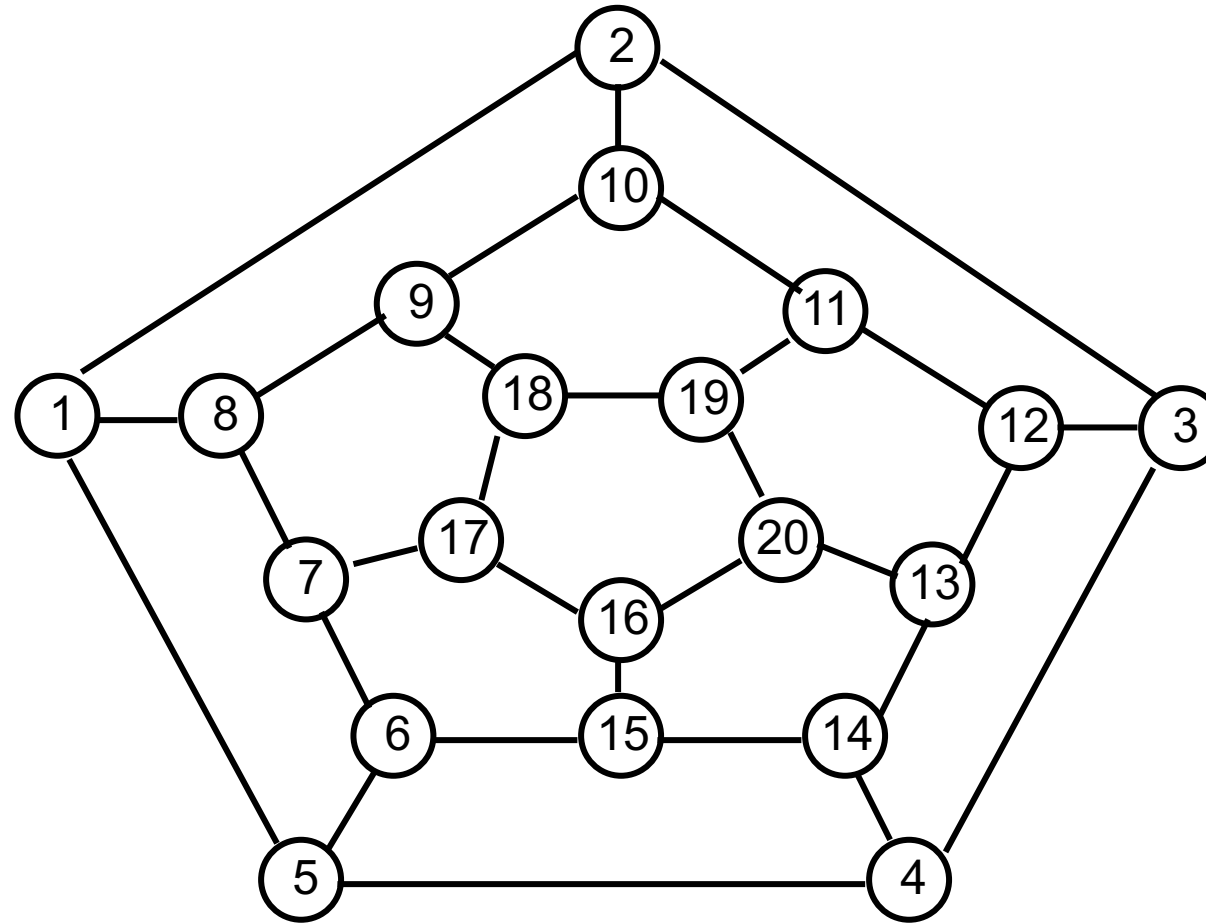


**G2**



# 作业

## 3. DFS, BFS 实现遍历 要求尽量按顶点序号顺序搜索 (作为练习)



## 实验六、图的构建与遍历

### 一、实验目的

1. 掌握图的**邻接矩阵**存储及实现方法
2. 掌握图的**邻接表**存储及实现方法
3. 掌握图的**建立**及**深度优先遍历**和**广度优先遍历**方法
3. 用C++语言实现相关算法，并上机调试。

### 二、实验内容

1. 实现图的邻接矩阵存储，并完成深度优先与广度优先遍历。
2. 实现图的邻接表存储，并完成深度优先与广度优先遍历。
3. 给出测试过程和测试结果。

**实验时间：** 第12周周四晚

**22网安：** 18:30-20:10    **22物联网：** 20:10-21:50

**实验地点：** 软件基础实验室301（老干部处）



*Thank You !*

*Q & A*