数据结构第六周笔记——图(上)(慕课浙大版本--XiaoYu)

6.1 什么是图

6.1.1 什么是图——定义

表示多对多的关系

包含

1. 一组顶点：通常用V(Vertex)表示顶点集合
2. 一组边：通常用E(Edge)表示边的集合
    1. 边是顶点对：(v,w)属于E，其中v,w属于V
    2. 有向边<v,w>表示从v指向w的边(单行线)
    3. 不考虑重边和自回路

抽象数据类型定义

1. 类型名称：图(Graph)
2. 数据对象集：G(V,E)由一个非空的有限顶点集合V和一个有限边集合E组成(可以一条边都没有，但不能一个顶点都没有)
3. 操作集：对于任意图G属于Graph，以及v属于V，e属于E
    1.
    ```
    1. Graph Create()：建立并返回空图
    2. Graph InsertVertex(Graph G, Vertex v)：将v插入G
    3. Graph InsertEdge(Graph G, Edge e)：将e插入G;
    4. void DFS(Graph G,Vertex v)：从顶点v出发深度优先遍历图G;
    5. void BFS(Graph G,Vertex v)：从顶点v出发宽度优先遍历图G;
    6. void ShortestPath(Graph G,Vertex v,int Dist[])：计算图G中顶点v到任意其他顶点的最短距离
    7. void MST(Graph G)：计算图G的最小生成树
    ```

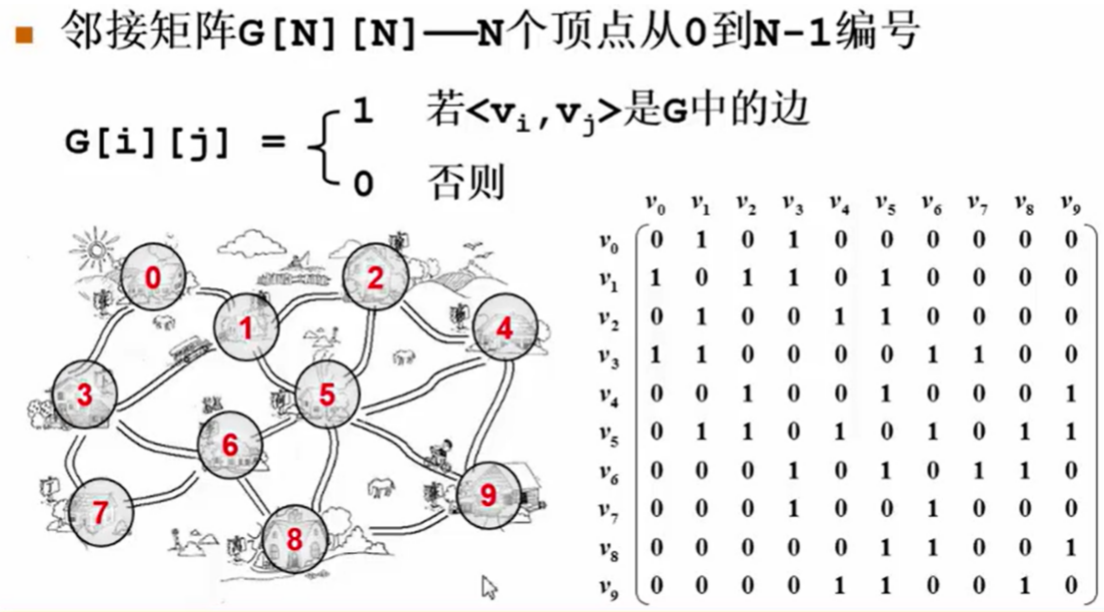
常见术语

1. 无向图：无所谓方向的
2. 有向图：在图中，若用箭头标明了边是有方向性(单向或者双向)的，则称这样的图为有向图，否则称为无向图。
3. 权重：边上显示的数字，可以有各种各样的现实意义
4. 网络：有带权重的图

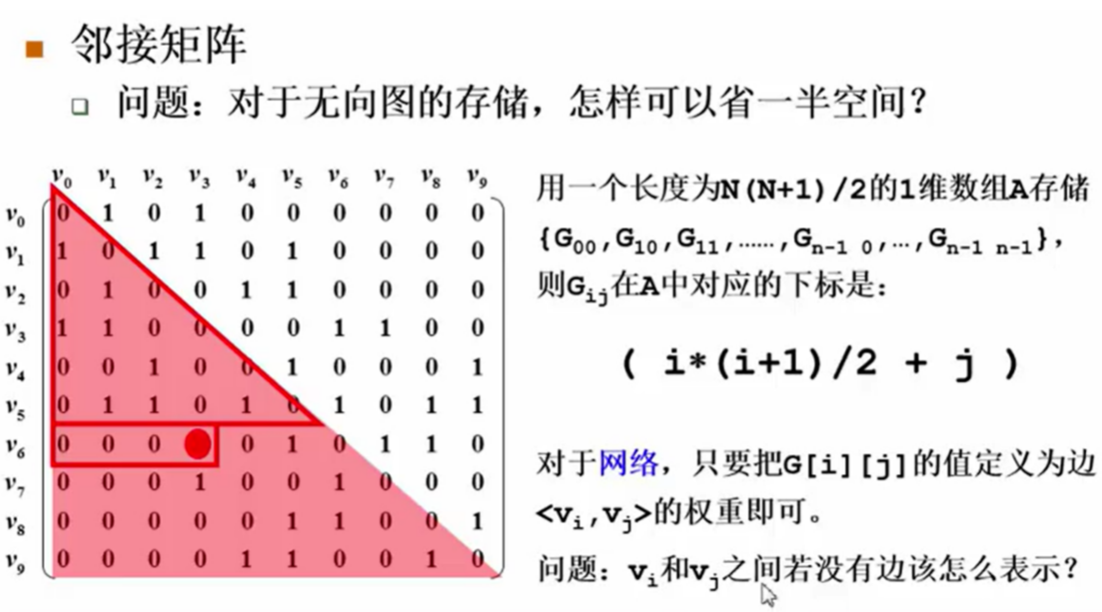
6.1.2 什么是图——邻接矩阵表示法

怎么在程序中表示一个图

1.



2.



邻接矩阵——有什么好处？

1. 直观、简单、好理解
2. 方便检查任意一对顶点间是否存在边
3. 方便找任一顶点的所有"邻接点"(有边直接相连的顶点)
4. 方便计算任一顶点的"度"(从该点出发的边数为"出度"，指向该点的边数为"入度")有向图的概念

邻接矩阵——有什么不好？

1. 浪费空间——存稀疏图(点很多而边很少)有大量无效元素

   但对稠密图(特别是完全图)还是很合算的

2. 浪费时间——统计稀疏图中一共有多少条边

无向图

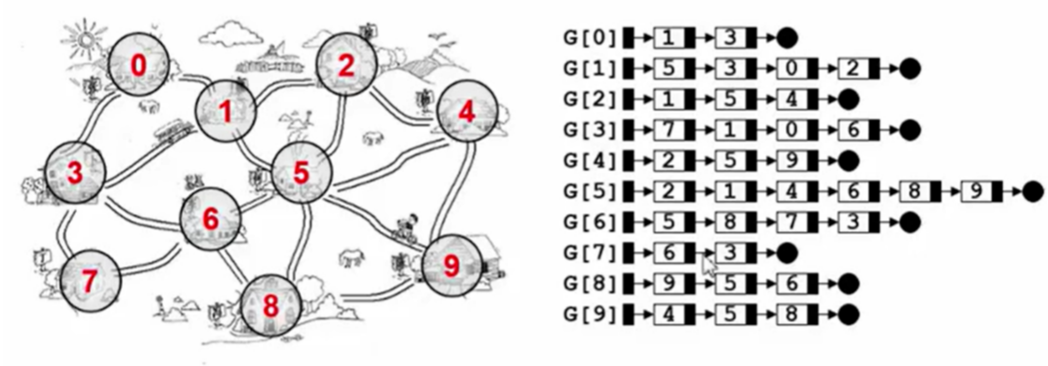
对应行(或列)非0元素的个数

有向图

对应行非0元素的个数是"出度";对应列非0元素的个数是"入度"

6.1.3 什么是图——邻接表表示法

邻接表：G[N]为指针数组，对应矩阵每行一个的链表，只存非0元素



上图的顺序是无所谓的，可以随意排列。使用这个表需要足够稀疏才合算

优点：

1. 方便找任一顶点的所有"邻接点"
2. 节约稀疏图的空间

   需要N个头指针+2E个结点(每个结点至少2个域)
3. 方便计算任一顶点的"度"

   对无向图：是的

   对有向边：只能计算"出度";需要构造"逆邻接表"(存指向自己的边)来方便计算"入度"
4. 方便检查任意一对顶点间是否存在边？NO

对于网络，结构中要增加权重的域

6.2 图的遍历

6.2.1 图的遍历——DFS

遍历：把图里面每个顶点都访问一遍而且不能有重复的访问

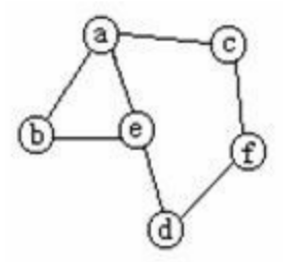
深度优先搜索(DFS)

当访问完了一个节点所有的灯后，一定原路返回对应着堆栈的出栈入栈的一个行为

深度优先搜索的算法描述

```
void DFS(Vertex V)//从迷宫的节点出来
{
    visited[V] = true;//给每个节点一个变量，true相当于灯亮了，false则是熄灭状态
    for(V的每个邻接点W)//视野看得到的灯
        if(!visited[W])//检测是否还有没点亮的
            DFS(W);//递归调用
}

//类似树的先序遍历
若有N个顶点、E条边，时间复杂度是
    用邻接表存储图，有O(N+E)//对每个点访问了一次，每条边也访问了一次
    用邻接矩阵存储图，有O(N²)//V对应的每个邻接点W都要访问一遍
```

1 已知一个图如下图所示，从顶点a出发按深度优先搜索法进行遍历，则可能得到的一种顶点序列为



A. a,e,b,c,f,d

B. a,b,e,c,d,f

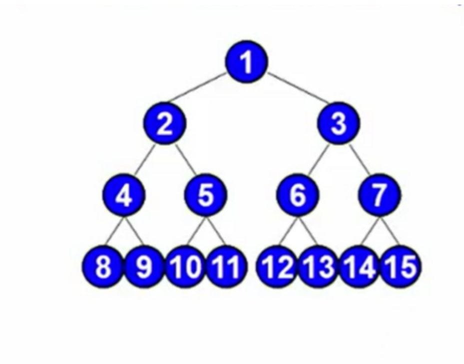
C. a,c,f,e,b,d

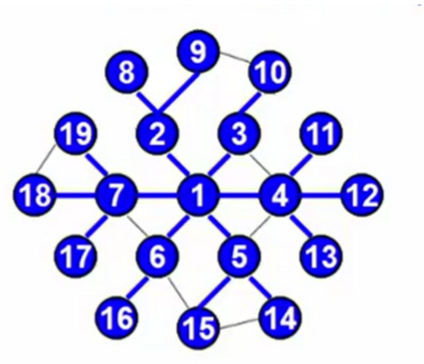
D. a,e,d,f,c,b

正确答案：D 你错选为B

6.2.2 图的遍历——BFS

广度优先搜索(Breadth First Search,BFS)





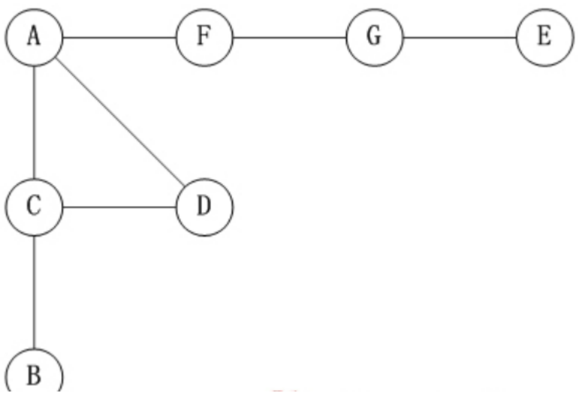
```
void BFS(Vertex V)
{
    visited[V] = true;
    Enqueue(V,Q);//压到队列里
    while(!IsEmpty(Q)){
        V = Dequeue(Q);//每次循环弹出一个节点
        for(V的每个邻接点W)
            if(!visited[W]){//没有访问过的去访问将其压入队列中
                visited[W] = true;
                Enqueue(W,Q);
            }
    }
}
```

若有N个顶点，E条边，时间复杂度是

用邻接表存储图，有O(N+E)

用邻接矩阵存储图，有$O(N^2)$

下面进行说明：



第1步：访问A。

第2步：访问(A的邻接点)C。在第1步访问A之后，接下来应该访问的是A的邻接点，即"C,D,F"中的一个。但在本文的实现中，顶点ABCDEFG是按照顺序存储，C在"D和F"的前面，因此，先访问C。

第3步：访问(C的邻接点)B。在第2步访问C之后，接下来应该访问C的邻接点，即"B和D"中一个(A已经被访问过，就不算在内)。而由于B在D之前，先访问B。

第4步：访问(C的邻接点)D。在第3步访问了C的邻接点B之后，B没有未被访问的邻接点；因此，返回到访问C的另一个邻接点D。

第5步：访问(A的邻接点)F。 前面已经访问了A，并且访问完了"A的邻接点B的所有邻接点(包括递归的邻接点在内)"；因此，此时返回到访问A的另一个邻接点F。

第6步：访问(F的邻接点)G。

第7步：访问(G的邻接点)E。

因此访问顺序是：A -> C -> B -> D -> F -> G -> E。

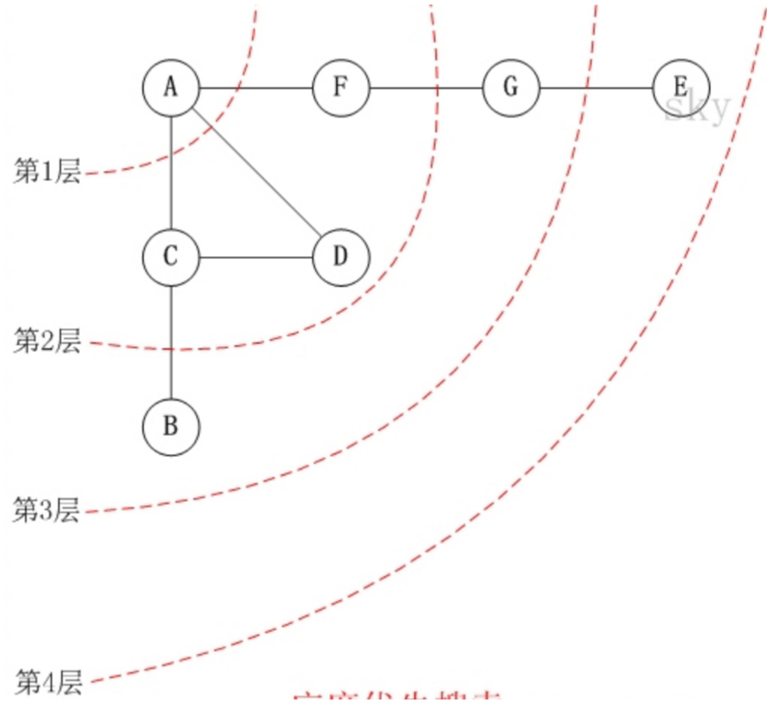
当然，上图是基于无向图，具体的代码在文章后面实现。

广度优先搜索

广度优先搜索算法(Breadth First Search)，又称为"宽度优先搜索"或"横向优先搜索"，简称BFS。

它的思想是：从图中某顶点v出发，在访问了v之后依次访问v的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得“先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止。

换句话说，广度优先搜索遍历图的过程是以v为起点，由近至远，依次访问和v有路径相通且路径长度为1,2...的顶点。



第1步：访问A。

第2步：依次访问C,D,F。在访问了A之后，接下来访问A的邻接点。前面已经说过，在本文实现中，顶点ABCDEFG按照顺序存储的，C在"D和F"的前面，因此，先访问C。再访问完C之后，再依次访问D,F。

第3步：依次访问B,G。在第2步访问完C,D,F之后，再依次访问它们的邻接点。首先访问C的邻接点B，再访问F的邻接点G。

第4步：访问E。在第3步访问完B,G之后，再依次访问它们的邻接点。只有G有邻接点E，因此访问G的邻接点E。

因此访问顺序是：A -> C -> D -> F -> B -> G -> E。

6.2.3 图的遍历——为什么需要两种遍历

在不同的情况下效率不同

广度跟深度的区别

1. 深度是直接一条路走到黑，碰壁没路走了在返回
2. 广度是一圈一圈的扫描过去，虽然前面还有路也不会强行深入

6.2.4 图的遍历——图不连通怎么办

连通：如果从V到W存在一条(无向)路径，则称V与W是连通的

路径：V到W路径是一系列顶点{V,v1,v2,....,vn,W}的集合，其中任一对相邻的顶点间都有图中的边。**路径的长度**是路径中的边数(如果带权(带权图)，则是所有边的权重和)。如果V到W之间的所有顶点都不同，则称为**简单路径**(有回路就不是简单路径)

回路：起点等于终点的路径

连通图：图中任意两顶点均连通

连通分量：无向图的极大连通子图

1. 极大顶点数：再加1个顶点就不连通了
2. 极大边数：包含子图中所有顶点相连的所有边

对有向图：

```
//强连通：有向图中顶点V和W之间存在双向路径，则称V和W是强连通的(路径可以不同同一条，但是一定是连通的)
//强连通图：有向图中任意两顶点均强连通
//强连通分量：有向图的极大强连通子图
//弱连通图：将强连通图的所有边的方向抹掉变成无向图就是连通的了
```

每调用一次DFS(V)，就把V所在的连通分量遍历了一遍,BFS也一样

```
void DFS(Vertex V)
{
    visited[V] = true;
    for(V的每个邻接点W)
        if(!visited[W])
            DFS(W);
}
```

遍历分量

```
void ListComponents(Graph G)
{
    for(each V in G)
        if(!visited[V]){
            DFS(V);//or BFS(V)
        }
}
```

6.3 应用实例：拯救007

```
void Save007(Graph G)
{
    for(each V in G){
        if(!visited[V] && FirstJump(V)){//这个FirstJump(V)是007第一跳有没有可能从孤岛跳到V上有没有可能，有且没踩过就跳上去
            answer = DFS(V);//or BFS(V)
            if(answer == YES) break;0
        }
    }
    if(answer == YES) output("Yes");
    else output("No");
}
```

DFS算法

```
void DFS(Vertex V)
{
    visited[V] = true;//表示鳄鱼头踩过了
    for(V的每个邻接点W)
        if(!visited[W])
            DFS(W);//递归
}
```

改良版本

```
void DFS(Vertex V)
{
    visited[V] = true;//表示鳄鱼头踩过了
    if(IOsSafe(V)) answer = YES;
    else{
    for(each W in G )
        if(!visited[W] && Jump(V,W)){//可以从V jump跳到这个W上面，作用是算V到W之间的距离是不是小于007可以跳跃最大距离
           answer = DFS(W);//递归
            if(answer == YES) break;
        }
    }
        return answer;
}
```

6.4 应用实例：六度空间(Six Degrees of Separation)

理论:

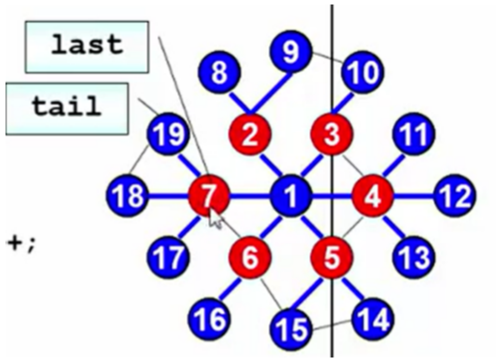
1. 你和任何一个陌生人之间所间隔的人不会超过6个
2. 给定社交网络图，请对每个节点计算符合"六度空间"理论的结点占结点总数的百分比

```
算法思路
1.对每个节点进行广度优先搜索
2.搜索过程中累计访问的节点数
3.需要记录"层"数,仅计算6层以内的节点数
```

```
void SDS()
{
    for(each V in G){
        count += BFS(V);
        Output = (count/N);
    }
}

//结合最初的BFS
    void BFS(Vertex V)
{
    visited[V] = true;count = 1;
    Enqueue(V,Q);//压到队列里
    while(!IsEmpty(Q)){
        V = Dequeue(Q);//每次循环弹出一个节点
        for(V的每个邻接点W)
            if(!visited[W]){//没有访问过的去访问将其压入队列中
                visited[W] = true;
                Enqueue(W,Q);count++;
            }
    }return count;
}
```

另外的解决方案

```
int BFS(Vertex V)
{
    vistex[V] = true;count = 1;
    level = 0;last = V;
    Enqueue(V,Q);
    while(!IsEmpty(Q)){
        V = Dequeue(Q);
        for( V的每个邻接点W)
            if(!visited[W]){
                visited[W] = true;
                Enqueue(W,Q);count++;
                tail = W;
            }
        if(V == last ){
            level++;last = tail;
        }
    }
    return count++;
}
```



小白专场：如何建立图：C语言实现

小白BG.1 邻接矩阵表示的图结点的结构

```
typedef struct GNode *PtrToGNode;//PtrToGNode是指向GNode的一个指针
struct GNode{
int Nv;//顶点数
int Ne;//边数
WeightType G[MaxVertexNum][MaxVertexNum];
    DataType Data[MaxVertexNum];//存顶点的数据
};
typedef PtrToGNode MGraph;//以邻接矩阵存储的图类型。定义为指向节点的指针。因为要用到的时候一个指针远远比一整个图来的快捷
```

小白BG.2 邻接矩阵表示的图——初始化

初始化一个有VertexNum个顶点但没有边的图

```
typedef int Vertex;//用顶点下标表示顶点，为整型
MGraph CreateGraph(int VertexNum)//VertexNum这个顶点数真的是整数，
{
    Vertex V , W;//我们在说V跟W的时候不是在说整数，而是顶点
    MGraph Graph;

    Graph = (MGraph)malloc(sizeof(struct GNode));
    Graph->Nv = VertexNum;
    Graph->Ne = 0;

    //注意：这里默认顶点编号从0开始，到(Graph->Nv - 1)
    for(V=0;V<Graph->Nv;V++)
        for((W=0;W<Graph->Nv;W++))
    Graph->G[V][M] = 0;//或者INFINITY，表示这两个顶点之间是没有边的

    return Graph
}
```

小白BG.3 邻接矩阵表示的图——插入边

```
typedef struct ENode *PtrToENode;
struct ENode{
    Vertex V1,V2;//有向边<V1,V2>，V1V2两个顶点一个出发点一个终点
    WeightType Weight;//权重，有权图才需要。权重的类型是WeightType
};
typedef PtrToENode Edge;

void InsertEdge(MGraph Graph,Edge E)
{
    //插入边<V1,V2>，这是一条边
    Graph->G[E->V1][E->V2] = E->Weight;

    //无向图的话还需要一条边(一共两条)，<V2,V1>
    Graph->G[E->V2][E->V1] = E->Weight;
}
```

小白BG.4 邻接矩阵表示的图——建立图

完整的建立一个MGraph

输入格式

1. Nv Ne
2. V1 V2 Weight
3. ……

```
MGraph BuildGraph()
{
    MGraph Graph;






    scanf("%d",&Nv);
    Graph = CreateGraph(Nv);
    //读入边数
    scanf("%d",&(Graph->Ne));
    if(Graph -> Ne = 0){//有边就还需要经过这里，没有边直接结束
        E = (Edge)malloc(sizeof(struct ENode));//临时存一下边
        for(i = 0; i < Graph->Ne; i++){
            scanf("%d %d %d",&E->V1,&E->V2,&E->Weight);
        InsertEdge(Graph,E);
        }
    }
    //如果顶点有数据的话，读入数据
    for(V=0;V<Graph->Nv;V++)
        scanf("%c",&(Graph->Data[V]));
    return Graph;
}
```

简易建法

```
int G[MAXN][MAXN],Nv,Ne;//声明为全局变量
void BuildGraph()
{
    int i,j,v1,v2,w;

    scanf("%d",&Nv);
    //CreateGraph
    for(i=0;i<Nv;i++)
        for(j=0;j<Nv;j++)
            G[i][j] = 0;//或INFINITY，把矩阵所有元素先初始化为0或者无穷大
    scanf("%d",&Ne);
    for(i = 0;i < Ne; i++){
        scanf("%d %d %d",&v1,&v2,&w);
        //InsertEdge
        G[v1][v2] = w;
        G[v2][v1] = w;
    }
}
```

小白BG.5 邻接表表示的图结点的结构

邻接表：G[N]为指针数组，对应矩阵每一行一个链表，只存非0元素

```
typedef struct GNode *PtrToGNode;
struct GNode {
    int Nv;//顶点数
    int Ne;//边数
    AdjList G;//邻接表
};
typedef PtrToGNode LGraph;
//以邻接表方式存储的图类型

//AdjList是自己定义的
typedef struct Vnode{
    PtrToAdjVNode FirstEdge;
    DataType Data;//存顶点的数据
}AdjList[MaxVertexNum];//AdjList是邻接表类型

typedef struct AdjVNode *PtrToAdjVNode;
struct AdjVNode{
    Vertex AdjV;//邻接点下标，定义为整型
    WeightType Weight;//边权重
    PtrToAdjVNode Next;
};
```

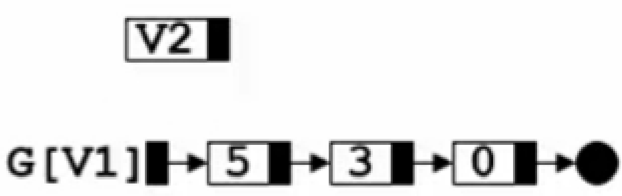
小白BG.6 邻接表表示的图——建立图

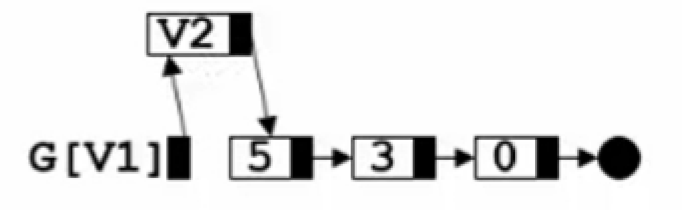
初始化一个有VertexNum个顶点但没有边的图

```
typedef int Vertex;//用顶点下标表示顶点，为整型
LGraph CreateGraph(int VertexNum)
{
    Vertex V,W;
    LGraph Graph;

    Graph = (LGraph)malloc(sizeof(struct GNode));
    Graph->Nv = VertexNum;
    Graph->Ne = 0;


    //没有边的意思是每个顶点跟着的那个链表都是空的
    //注意：这里默认顶点编号从0开始，到(Graph->Nv - 1)
    for(V=0;V<Graph->Nv;V++)
    Graph->G[V].FirstEdge = NULL;

    return Graph;
}
```

向LGraph中插入边

```
void InsertEdge(LGraph Graph,Edge E)
{
    PtrToAdjVNode NewNode;
    //-------------------插入边<V1,V2>------------------------------
    //为V2建立新的邻接点
    NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjNode));
    NewNode->AdjV = E->V2;
    NewNode->Weight = E->Weight
        //将V2插入到V1的表头
        NewNode->Next = Graph->G[E->V1].FirstEdge;
    Graph->G[E->V1].FirstEdge = NewNode;

    //-------------------若是无向图，还需插入边<V2,V1>------------------
        //为V1建立新的邻接点
    NewNode = (PtrToAdjVNode)malloc(sizeof(struct AdjNode));
    NewNode->AdjV = E->V1;
    NewNode->Weight = E->Weight
        //将V2插入到V1的表头
        NewNode->Next = Graph->G[E->V2].FirstEdge;
    Graph->G[E->V2].FirstEdge = NewNode;
}
```





完整建立一个LGraph

```
LGraph BuildGraph()
{
    LGraph Graph;
    ...............
}
```