图

#图的存储结构

(1)无向图的邻接表
由于无向图每一条边没有确定的方向，第i行第j列表示第i个元素和第j个元素之间有一条边。

（2）有向图的邻接矩阵
矩阵中第i行表示从第i个元素出发的边，其中的非零元数量表示的是第i个元素的出度；
矩阵中第j列表示进入第j个元素的边，其中的非零元数量为第j个元素的入度。
某个元素的总度数为元素的出度 + 入度之和。

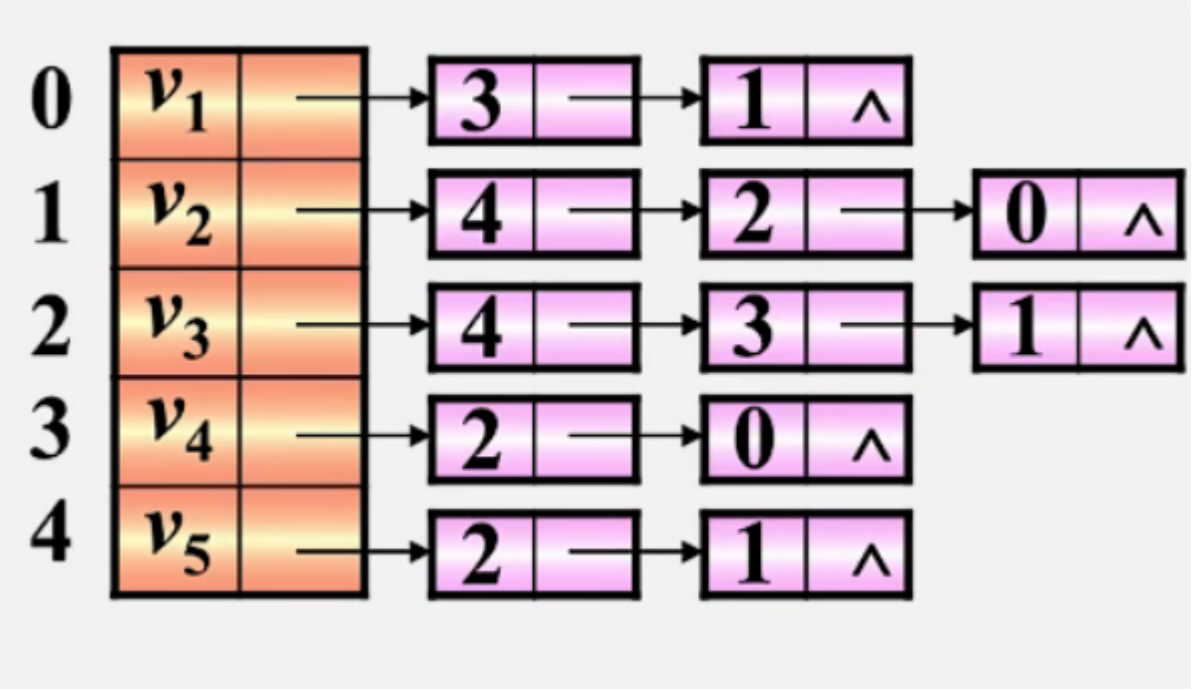
（3）邻接矩阵的优缺点：

- 优点：方便操作，容易判断两点之间是否有边的关系，方便查找所有点的度数和邻接点
- 缺点：插入、删除、修改一个点很麻烦，采用邻接矩阵存储图时需要消耗大量空间，对于稀疏图会浪费很多

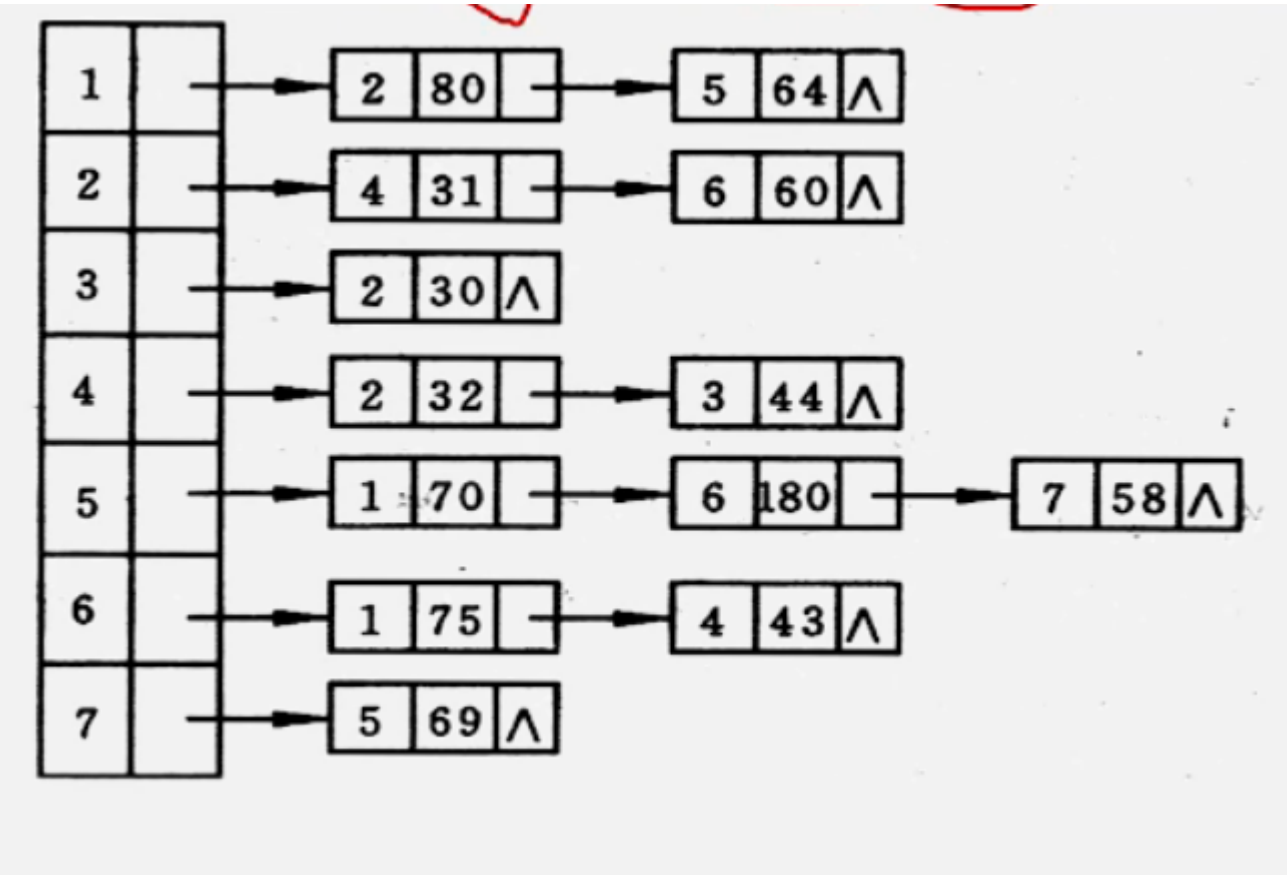
（3）邻接表法
使用一个线性表来存储所有节点，每个节点的元素有：本身对应的权值，对应的第一条边的节点，这个节点之后对应着第二条边的节点。

使用的是线性链表。每个头表之后连上边表。边链表最多可有三个元素，第一个是由点出来之后链接的节点标号，第二个是其他节点，此外还存放这条边的权值。

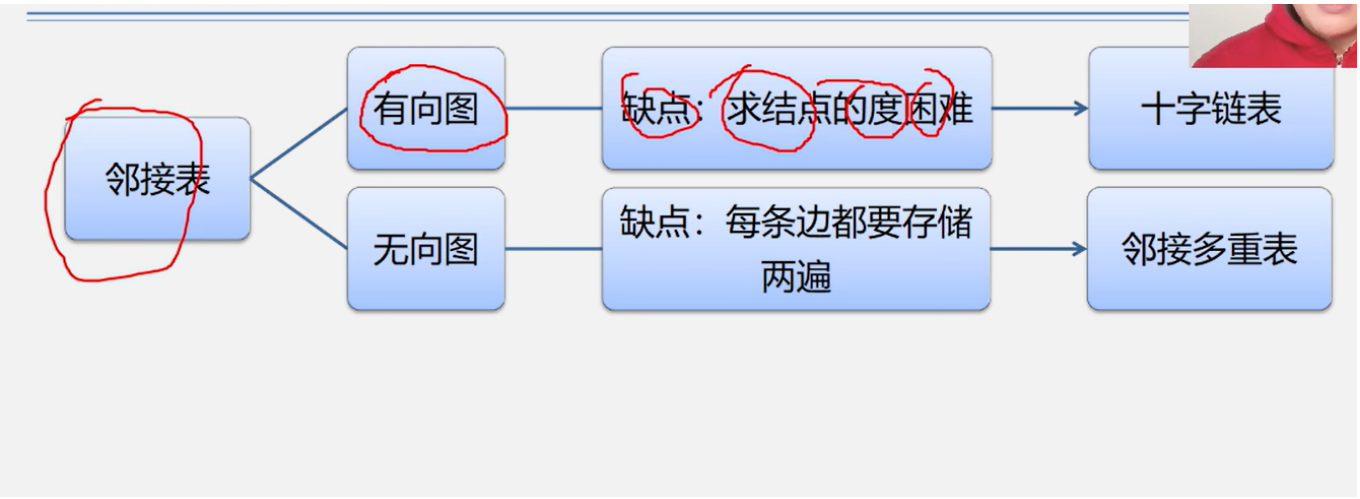
空间复杂度是o(n+2* e);



后面的紫色部分即为边链表，前面的是点链表。
有向的存储方式和无向一样。



邻接表法的缺点：对于有向邻接表，不方便计算节点的度；对于无向邻接表，每条边要存储两边，浪费



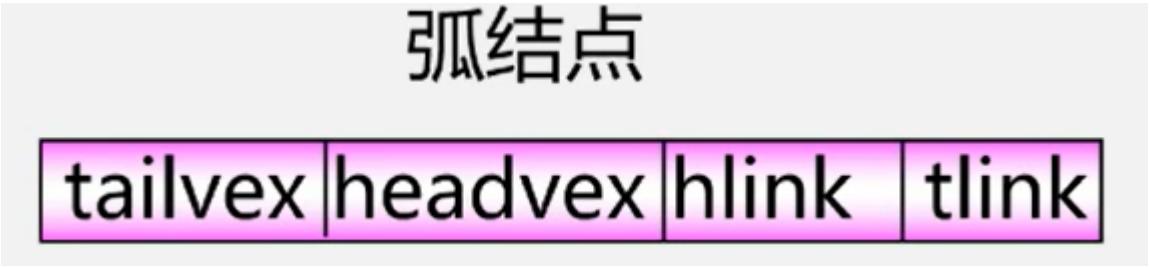
对于每个确定的图，他的邻接矩阵是一定的，他的邻接表是不一定的

（4）十字链表
为了存储无向图，尽量的节省空间并方便统计节点的度数，可用十字链表。

点链表变为：

firstout 用于指向点的第一个出边链接的节点，并一直指到最后一个出边的节点；
firstin同理。

边链表变为：



其中hlink指向一条和他共享一个头节点的边，tlink指向一个和他共享尾结点的边

（5）邻接多重表
较为复杂，不会考察。

（6）dfs遍历图
邻接矩阵复杂度是n^2，邻接表是n+e；

（7）bfs遍历图
结果与bfs是一样的，只不过用队列来实现

#MST算法
取图中的一个点，则存在点集u与v,其中u用于存放在最小生成树的点，v是不存在于生成树但是在全集中的点，则取这两个点的边的最小权值边加入树中即可得到最小生成树。

#floyd算法
三重循环，floyd的作用是寻找任意两点之间的最短路径。
原题：
854. Floyd求最短路 - AcWing题库

```
# include<bits/stdc++.h>
# include<algorithm>
typedef long long ll;
const int inf=1e9;

using namespace std;
```

```
ll n,m,q;
ll p[2030][2030];
ll dis[2030];
void floyd()
{
        for(int k=1;k<=n;k++)
        for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
        p[i][j]=min(p[i][j],p[i][k]+p[k][j]);

}
int main (void)
{
        cin.tie(0);
        cout.tie(0);
        cin>>n>>m>>q;
        for(int i=1;i<=n;i++)
        for(int j=1;j<=n;j++)
        if(i==j)
        p[i][j]=0;
        else
        p[i][j]=inf;

        for(int i=0;i<m;i++)
        {
                ll a,b,c;
                cin>>a>>b>>c;
                p[a][b]=min(p[a][b],c);
                }
        floyd();
        while(q--)
        {
                int a,b;
                cin>>a>>b;
                int t=p[a][b];
                if(p[a][b]>inf/2)
                cout<<"impossible"<<endl;
                else
                cout<<p[a][b]<<endl;

        }
        return 0;
```

```
}
```

#dijkstra算法

dijkstra算法用于寻找某个特定顶点到其他点的最短距离。
提供一种实现方法：

```
# include<bits/stdc++.h>
# include<algorithm>
typedef long long ll;
using namespace std;
int n,m;
bool istrue[114514];
int p[510][510];
int dis[510];
int dijstra()
{
        memset(dis,0x3f,sizeof dis);
        dis[1]=0;

        for(int i=0;i<n;i++)
        {
                int t=-1;
                for(int j=1;j<=n;j++)
                {
                        if(!istrue[j]&&(t==-1||dis[t]>dis[j]))
                                t=j;
                }
                for(int i=1;i<=n;i++)
                {
                        dis[i]=min(dis[i],dis[t]+p[t][i]);

                }
                istrue[t]=true;

        }
        if(dis[n]==0x3f3f3f3f)
        return -1;
        else
        return dis[n];
}

int main (void)
```

```
{
        cin.tie(0);
        cout.tie(0);
        cin>>n>>m;
        memset(p,0x3f,sizeof p);
        for(int i=0;i<m;i++)
        {
                int a,b,c;
                cin>>a>>b>>c;
                p[a][b]=min(p[a][b],c);

        }
        int t=dijstra();
        cout<<t<<endl;
        return 0;

}
```

但是很遗憾，这种算法只能处理1e5以下的边数。为此可以把dijkstra算法进一步优化，产生优化dijkstra算法：

```
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 210, INF = 1e9;

int n, m, Q;
int d[N][N];

void floyd()
{
    for (int k = 1; k <= n; k ++ )
        for (int i = 1; i <= n; i ++ )
            for (int j = 1; j <= n; j ++ )
                d[i][j] = min(d[i][j], d[i][k] + d[k][j]);
}

int main()
{
```

```
    scanf("%d%d%d", &n, &m, &Q);

    for (int i = 1; i <= n; i ++ )
        for (int j = 1; j <= n; j ++ )
            if (i == j) d[i][j] = 0;
            else d[i][j] = INF;

    while (m -- )
    {
        int a, b, c;
        scanf("%d%d%d", &a, &b, &c);
        d[a][b] = min(d[a][b], c);
    }

    floyd();

    while (Q -- )
    {
        int a, b;
        scanf("%d%d", &a, &b);

        int t = d[a][b];
        if (t > INF / 2) puts("impossible");
        else printf("%d\n", t);
    }

    return 0;
}
```

#拓扑序列

有向无环图才有拓扑序列。

拓扑序列是一种对某个图的顶点的排序手段，使得在有向图中的所有边的起点排在终点的前面。

```
#include <cstring>
#include <iostream>
#include <algorithm>

using namespace std;

const int N = 100010;

int n, m;
int h[N], e[N], ne[N], idx;
```

```
int d[N];
int q[N];

void add(int a, int b)
{
    e[idx] = b, ne[idx] = h[a], h[a] = idx ++ ;
}

bool topsort()
{
    int hh = 0, tt = -1;

    for (int i = 1; i <= n; i ++ )
        if (!d[i])
            q[ ++ tt] = i;

    while (hh <= tt)
    {
        int t = q[hh ++ ];

        for (int i = h[t]; i != -1; i = ne[i])
        {
            int j = e[i];
            if (-- d[j] == 0)
                q[ ++ tt] = j;
        }
    }

    return tt == n - 1;
}

int main()
{
    scanf("%d%d", &n, &m);

    memset(h, -1, sizeof h);

    for (int i = 0; i < m; i ++ )
    {
        int a, b;
        scanf("%d%d", &a, &b);
        add(a, b);
```

```
        d[b] ++ ;
    }

    if (!topsort()) puts("-1");
    else
    {
        for (int i = 0; i < n; i ++ ) printf("%d ", q[i]);
        puts("");
    }

    return 0;
}
```

(5条消息) 还不会拓扑排序？看这一篇就够了_lareges的博客-CSDN博客

因此有向无环图也被称为拓扑图。