```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////
//   用邻接表表示图的类实现，包括顶点结点的类/结构实现、边结点类/结构定义和图类实现
//   Author：Melissa M.CAO
//   Belong：Section of software theory，School of Computer Engineering & Science，Shanghai University
//   Version： 1.0
////////////////////////////////////////////////////////////////////////////////
//////////////////////////

#include "StdAfx.h"
#include "AdjacencyListGraph.h"
#include "MinSpanTree.h" //最小生成树算法中用到
//#include "LinkQueue.cpp"

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//  构造函数
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> AdjacencyListGraph<vertexType, arcType>::AdjacencyListGraph(int type1, int type2)
{
    if (typ1 != 1 && type1 != 2)
    {
        cout << "既不是无向图又不是有向图，无所适从，不干了，返回！" << endl;
        return;
    }

    if (typ2 != 1 && type2 != 2)
    {
        cout << "既不是带权图又不是无权图，无所适从，不干了，返回！" << endl;
        return;
    }

    graphType = type1;
    weightGraph = type2;
    CurrentNumVertexes = 0;
    CurrentNumArcs = 0;
    edgeMaxValue = 10000000;
}

////////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////
//  构造函数
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> AdjacencyListGraph<vertexType,
arcType>::AdjacencyListGraph (vertexType v[ ] , int num, int type1, int type2)
{
    int i;

    VertexNode<vertexType, arcType> temp;
//  VertexesTable.Initial();
    for (i = 0 ;i < num; i++)
    {
        temp.data = v[i];
        temp.firstarc = NULL;
        VertexesTable.AppendItem(temp);
    }

    graphType = type1;
    weightGraph = type2;
    CurrentNumVertexes = num;
    CurrentNumArcs = 0;
    edgeMaxValue = 10000000;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//  析构函数
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> AdjacencyListGraph<vertexType,
arcType>::~AdjacencyListGraph( )
{
    int i;
    ArcNode<arcType> *p;
    for (i = 0; i < CurrentNumVertexes; i++)
    {
        p = VertexesTable.data[i].firstarc;
        while ( p != NULL)
        {
            VertexesTable.data[i].firstarc = p->nextarc;
            delete p;
            p = VertexesTable.data[i].firstarc;
        }
```

```
    }
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//显示图的基本信息
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::Display()
{
    if (graphType ==1)
        cout << "您所建立的图是无向图，";
    else
        if (graphType ==2)
            cout << "您所建立的图是有向图，";
        else {
            cout <<"您所建立的图既不是无向图也不是有向图，不知道哪里出错了！" <<
endl;
            return;
        }

    if (weightGraph ==1)
        cout << "该图无权值，";
    else
        if (weightGraph ==2)
            cout << "该图有权值，";
        else {
            cout << "该图既不是无权值也不是有权值，不知道哪里出错了！" << endl;
            return;
        }

    cout << "该图具有" << CurrentNumVertexes << "个顶点，有" << CurrentNumArcs <<
"条边" << endl;
    cout << "该图的邻接表为：" << endl;
    int i;
    VertexNode<vertexType, arcType> temp;
    ArcNode<arcType> *edge;
    for (i = 0; i < CurrentNumVertexes; i++)
    {
        temp = VertexesTable.GetData(i);
        cout << temp.data << "---->";
        if (temp.firstarc == NULL)
            cout << "NULL" << endl;
```

```
        else //访问链表
        {
            edge = temp.firstarc;
            while (edge != NULL)
            {
                cout << "<" << (VertexesTable.GetData(edge->adjvex)).data;
                if (weightGraph==2)
                    cout << ", " << edge->weight;
                cout << "> ---->";
                edge = edge->nextarc;
            }
            cout << "NULL" << endl;
        } //end of else 访问链表
    } // end of for
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//根据结点的下标序号，返回结点的值，序号从0开始，为顶点在顺序表中的位置
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> vertexType
AdjacencyListGraph<vertexType,arcType>::GetValue(int v)
{
    if (v >= 0 && v < CurrentNumVertexes)
        return VertexesTable.data[v].data;
    else
        return NULL;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
// 取顶点v在数组中的位置
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> int
AdjacencyListGraph<vertexType,arcType>::GetVertexPos( const vertexType &v )
{
    int i;

    for (i = 0; i < CurrentNumVertexes; i++)
        if (VertexesTable.data[i].data == v)
            break;
    if (i >= CurrentNumVertexes)
```

```
        i = -1;

    return i;
};

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//根据两个顶点的下标序号，返回该两顶点之间狐的权值，序号从0开始，为顶点在顺序表中的位置
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> arcType AdjacencyListGraph<vertexType,arcType>::GetWeight ( int v1, int v2 )
{
    if (weightGraph != 2)
    {
        cout << "该图不是带权图，图中所有的边都没有权值！";
        return NULL;
    }
    if (v1 < 0 || v1 >= CurrentNumVertexes || v2 < 0 || v2 >= CurrentNumVertexes)
    {
        cout << "您查找的边的起点或终点不存在，无法给出权值！" << endl;
        return NULL;
    }

    if (v1 == v2)
    {
        cout << "要查找的边的始点或终点相同，自身间的边不考虑！" << endl;
        return NULL;
    }
    //
    ArcNode<arcType> *p;
    p = VertexesTable.data[v1].firstarc;
    while (p != NULL)
        if (p->adjvex == v2)
            return p->weight;
        else
            p = p->nextarc;
    return NULL;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//根据两个顶点的值，返回该两顶点之间弧的权值，序号从0开始，为顶点在顺序表中的位
```

置
```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> arcType AdjacencyListGraph<vertexType,arcType>::GetWeight ( vertexType v1, vertexType v2 )
{
    if (weightGraph == 1)
    {
        cout << "该图不带权值，您查找的边无法给出权值！" << endl;
        return NULL;
    }
    int begin, end;
    begin = GetVertexPos(v1);
    end = GetVertexPos(v2);
    if (begin < 0 || begin >= CurrentNumVertexes || end < 0 || end >= CurrentNumVertexes)
    {
        cout << "您查找的边的起点或终点不存在，无法给出权值！" << end;
        return NULL;
    }

    if (begin == end)
    {
        cout << "要查找的边的始点或终点相同，自身间的边不考虑！" << endl;
        return NULL;
    }
    //
    return GetWeight(begin, end);
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
// 插入边，给出两个顶点的序号，和该两顶点之间狐的权值，序号从 0 开始，为顶点在顺序表中的位置
//参数：v1--顶点 1 的序号，v2--顶点 2 的序号,w--边上的权值，insertpos--插入位置，1 表示插在边链表的尾部，2 表示插在边链表的头部
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,arcType>::InsertArc(int v1,int v2, arcType w, int insertpos)
{
    if (v1 < 0 || v1 >= CurrentNumVertexes || v2 < 0 || v2 >= CurrentNumVertexes)
    {
```

```
        cout << "您插入的边的起点或终点不存在，无法进行插入！" << endl;
        return;
    }

    if (v1 == v2)
    {
        cout << "要插入的边的始点或终点相同，自身间的边不考虑！" << endl;
        return;
    }
    //
    ArcNode<arcType> *p = new ArcNode<arcType>(v2, w);
    if (insertpos == 2 || VertexesTable.data[v1].firstarc == NULL)
    {
        p->nextarc = VertexesTable.data[v1].firstarc;
        VertexesTable.data[v1].firstarc = p;
        CurrentNumArcs++;
    }
    else
    {
        ArcNode<arcType> *q = VertexesTable.data[v1].firstarc;
        while (q->nextarc != NULL)
            q = q->nextarc;
        q->nextarc = p;
        CurrentNumArcs++;
    }

    //如果是无向图，同时在 v2 的边链表中插入
    if (graphType == 1)
    {
         ArcNode<arcType> *p1 = new ArcNode<arcType>(v1, w);
        if (insertpos == 2 || VertexesTable.data[v2].firstarc == NULL)
        {
            p1->nextarc = VertexesTable.data[v2].firstarc;
            VertexesTable.data[v2].firstarc = p1;
        }
        else
        {
            ArcNode<arcType> *q = VertexesTable.data[v2].firstarc;
            while (q->nextarc != NULL)
                q = q->nextarc;
            q->nextarc = p1;
        }
    }
}
```

```
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
// 插入边，给出两个顶点的值，和该两顶点之间弧的权值，序号从 0 开始，为顶点在顺序表中的位置
//参数：v1--顶点 1 的值，v2--顶点 2 的值,w--边上的权值, insertpos--插入位置，1 表示插在边链表的尾部，2 表示插在边链表的头部
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,arcType>::InsertArc(vertexType v1, vertexType v2, arcType w, int insertpos)
{
    int begin, end;
    begin = GetVertexPos(v1);
    end = GetVertexPos(v2);
    InsertArc(begin, end, w, insertpos);
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
// 插入边，给出两个顶点的序号，和该两顶点之间弧的权值，序号从 0 开始，为顶点在顺序表中的位置
//参数：v1--顶点 1 的序号，v2--顶点 2 的序号,w--边上的权值，insertpos--插入位置，1 表示插在边链表的尾部，2 表示插在边链表的头部
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,arcType>::InsertArc(int v1,int v2, int insertpos)
{

    if (v1 < 0 || v1 >= CurrentNumVertexes || v2 < 0 || v2 >= CurrentNumVertexes)
    {
        cout << "您插入的边的起点或终点不存在，无法进行插入！" << endl;
        return;
    }

    if (v1 == v2)
    {
        cout << "要插入的边的始点或终点相同，自身间的边不考虑！" << endl;
        return;
    }
    //
    ArcNode<arcType> *p = new ArcNode<arcType>(v2);
```

```
    if (insertpos == 2 || VertexesTable.data[v1].firstarc == NULL)
    {
        p->nextarc = VertexesTable.data[v1].firstarc;
        VertexesTable.data[v1].firstarc = p;
        CurrentNumArcs++;
    }
    else
    {
        ArcNode<arcType> *q = VertexesTable.data[v1].firstarc;
        while (q->nextarc != NULL)
            q = q->nextarc;
        q->nextarc = p;
        CurrentNumArcs++;
    }

    //如果是无向图，同时在 v2 的边链表中插入
    if (graphType == 1)
    {
        ArcNode<arcType> *p1 = new ArcNode<arcType>(v1);
        if (insertpos == 2 || VertexesTable.data[v2].firstarc == NULL)
        {
            p1->nextarc = VertexesTable.data[v2].firstarc;
            VertexesTable.data[v2].firstarc = p1;
        }
        else
        {
            ArcNode<arcType> *q = VertexesTable.data[v2].firstarc;
            while (q->nextarc != NULL)
                q = q->nextarc;
            q->nextarc = p1;
        }
    }
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
// 插入边，给出两个顶点的值，和该两顶点之间弧的权值，序号从 0 开始，为顶点在顺序表中的位置
//参数：v1--顶点 1 的值，v2--顶点 2 的值,w--边上的权值, insertpos--插入位置，1 表示插在边链表的尾部，2 表示插在边链表的头部
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,arcType>::InsertArc(vertexType v1,vertexType v2,
```

```
int insertpos)
{
    int begin, end;
    begin = GetVertexPos(v1);
    end = GetVertexPos(v2);
    InsertArc(begin, end, insertpos);
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
// 给出顶点的序号，查找该顶点邻接的第一条边，序号从 0 开始，为顶点在顺序表中的位置
//参数：v1--顶点 1 的值；返回：第一条邻接边的另一端顶点序号信息
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> int
AdjacencyListGraph<vertexType,arcType>::GetFirstNeighbor ( int v )
{
    if (v >= 0 && v < CurrentNumVertexes)
    {
        ArcNode<arcType> *p = VertexesTable.data[v].firstarc;
        if (p != NULL)
            return p->adjvex;
    }
    return -1;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
// 给出两顶点的序号,查找第一个顶点所邻接的在第二个顶点之后的一条边,序号从 0 开始，
为顶点在顺序表中的位置
//参数：v1--顶点 1 的序号；v2--顶点 2 的序号；返回：下一条邻接边的另一端顶点序号信
息
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> int
AdjacencyListGraph<vertexType,arcType>::GetNextNeighbor ( int v1,  int v2 )
{
    if (v1 < 0 || v1 >= CurrentNumVertexes || v2 < 0 || v2 >= CurrentNumVertexes)
    {
        cout << "起点或终点不存在，无法进行查找吓一条边！" << endl;
        return -1;
    }

    ArcNode<arcType> *p = VertexesTable.data[v1].firstarc;
```

```
    while(p != NULL)
    {
        if (p->adjvex == v2 && p->nextarc != NULL)
            return p->nextarc->adjvex;
        else
            p = p->nextarc;
    }

    return -1;
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
// 删除边，给出两个顶点的序号，和序号从 0 开始，为顶点在顺序表中的位置
//参数：v1--顶点 1 的序号，v2--顶点 2 的序号；
//删除<v1，v2>
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void
AdjacencyListGraph<vertexType,arcType>::DeleteArc ( int v1, int v2 )
{
    if (v1 < 0 || v1 >= CurrentNumVertexes || v2 < 0 || v2 >= CurrentNumVertexes)
    {
        cout << "您删除的边的起点或终点不存在，无法进行删除！" << endl;
        return;
    }

    if (v1 == v2)
    {
        cout << "要删除的边的始点或终点相同，自身间的边不考虑！" << endl;
        return;
    }

    //
    ArcNode<arcType> *p = VertexesTable.data[v1].firstarc;
    if (p == NULL)
        return;
    if (p->adjvex == v2)
    {
        VertexesTable.data[v1].firstarc = p->nextarc;
        delete p;
        CurrentNumArcs--;
    }
    else
```

```
    {
        while (p->nextarc != NULL && p->nextarc->adjvex != v2)
            p = p->nextarc;
        if (p->nextarc != NULL)
        {
            ArcNode<arcType> *q = p->nextarc;
            p->nextarc = q->nextarc;
            delete q;
            CurrentNumArcs--;
        }
    }

    //如果是无向图，同时在 v2 的边链表中删除
    if (graphType == 1)
    {
        p = VertexesTable.data[v2].firstarc;
        if (p == NULL)
        {
            cout << "无向图的邻接表不对称，程序可能出错了！" << endl;
            exit(1);
        }
        if (p->adjvex == v1)
        {
            VertexesTable.data[v2].firstarc = p->nextarc;
            delete p;
        }
        else
        {
            while (p->nextarc != NULL && p->nextarc->adjvex != v1)
                p = p->nextarc;
            if (p->nextarc != NULL)
            {
                ArcNode<arcType> *q = p->nextarc;
                p->nextarc = q->nextarc;
                delete q;
            }
        } // end of else
    } //end of if (graphType == 1)
}

////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
// 删除顶点，序号从 0 开始，为顶点在顺序表中的位置；参数：v1--顶点 1 的序号；
////////////////////////////////////////////////////////////////////////////
```

```
////////////////////////////////////////////////
template<class vertexType, class arcType> void
AdjacencyListGraph<vertexType,arcType>::DeleteVertex ( int v )
{
    if (v < 0 || v >= CurrentNumVertexes)
    {
        cout << "您删除的顶点不存在，无法进行删除！" << endl;
        return;
    }
    //先删除 v 的邻接表
    ArcNode<arcType> *p = VertexesTable.data[v].firstarc;
    while (p !=NULL)
    {
        DeleteArc(v, p->adjvex);
        p = VertexesTable.data[v].firstarc;
    }
    //VertexesTable.Delete(v, true);
    // 问题：需要修改邻接表中每一条边结点的 adjvex 值；
    int i;
    for (i = 0; i< CurrentNumVertexes; i++)
    {
        p = VertexesTable.data[i].firstarc;
        while (p != NULL)
        {
            if (p->adjvex > v)
                p->adjvex--;
            p = p->nextarc;
        }
    }

    //删除结点
    CurrentNumVertexes--;
    for (i = v; i< CurrentNumVertexes; i++)
        VertexesTable.data[i] = VertexesTable.data[i+1];

}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
// 插入顶点，序号从 0 开始，为顶点在顺序表中的位置；参数：v1--顶点 1 的序号；
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> int
AdjacencyListGraph<vertexType,arcType>::InsertVertex ( vertexType & v )
```

```
{
    VertexNode<vertexType, arcType> temp;
    temp.data = v;
    temp.firstarc = NULL;
    CurrentNumVertexes++;
    return VertexesTable.AppendItem(temp);
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
// 获得指向序号 v 的结点的第一邻接边的指针
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> ArcNode<arcType>*
AdjacencyListGraph<vertexType,arcType>::GetAdj(int v)
{
    if (v < 0 || v >= CurrentNumVertexes)
    {
        cout <<"您指定的顶点不存在，无法找到指向它的第一条邻接边的指针！"<< endl;
        return NULL;
    }

    ArcNode<arcType> *p = VertexesTable.data[v].firstarc;
    return p;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
// 获得指向序号 v 的结点的邻接边(v,u)的指针
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> ArcNode<arcType>*
AdjacencyListGraph<vertexType,arcType>::GetAdj(int v, int u)
{
    if (v < 0 || v >= CurrentNumVertexes || u < 0 || u >= CurrentNumVertexes)
    {
        cout << "您指定的顶点不存在，无法找到指向它的邻接边的指针！" << endl;
        return NULL;
    }

    ArcNode<arcType> *p = NULL;
    for (p = VertexesTable.data[v].firstarc; p; p = p->nextarc)
        if (p->adjvex == u)
            break;
```

```
    return p;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//判断图中边的权值是否含有负数
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> bool
AdjacencyListGraph<vertexType,arcType>::HavePostiveEdge()
{
    int n = NumberOfVertexes();
    int i;
    ArcNode<arcType> *p = NULL;

    for (i = 0; i < n; i++)
    {
        for (p = VertexesTable.data[i].firstarc; p; p = p->nextarc)
            if (p->weight < 0)
                return false;
    }

    return true;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//-------------------------------------------------------------------扩展操作
-----------------------------------------
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//定义的抽象的“访问/遍历”函数
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::Visit( vertexType v )
{
    int i = GetVertexPos(v);

    if (i < 0 || i >= NumberOfVertexes())
    {
        cout << "要访问/遍历的顶点不存在" << endl;
        return;
    }
```

```
    else
        cout << "顶点值“" << v << "”是图中的第" << i+1 << "个顶点！" << endl;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//定义的抽象的“访问/遍历”函数
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::Visit( vertexType v, int mode, int *visited )
{
    int i = GetVertexPos(v);
    int all =  NumberOfVertexes();

    if (i < 0 || i >= all)
    {
        cout << "要访问/遍历的顶点不存在" << endl;
        return;
    }
    else
    {
        switch (mode)
        {
            case 1: //一般的访问，输出即可
                cout << "顶点值“" << v << "”是图中的第" << i+1 << "个顶点！" <<
endl;
            break;

            case 2: //相应的 visited 上记录输出顺序
                cout << "顶点值“" << v << "”是图中的第" << i+1 << "个顶点！" <<
"但却是第" << visited[i] << "个被输出的结点！" << endl;
            break;

            case 3: //相应的 visited 上记录输出顺序
                cout << "原序号为“" << i+1 << "”的顶点是“" << v << "”, 其新的
编号为“" << all - visited[i] + 1 << "”！" << endl;
            break;
        }

    }
}

//////////////////////////////////////////////////////////////////////////////
```

```
//////////////////////////////////////////////
//深度优先搜索--递归算法
//   参数： v,指定顶点的下标;*visited,记录顶点访问记录,即每个顶点是否被访问过;
//          count：用于记录在遍历中已经访问了多少个顶点。
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::DFS(const int v, int *visited, int *count)
{
    *count = *count + 1;
    vertexType temp = GetValue (v);
    Visit(temp);  //访问顶点 v, 例如 cout, modify
    visited[v] = 1;          //顶点 v 作访问标记

    int w = GetFirstNeighbor (v);   //w 是返回的顶点的下标
    while ( w != -1 )         //若顶点 w 存在
    {
        if ( !visited[w] )
            DFS (w, visited, count);
        w = GetNextNeighbor (v, w);
    }  //重复检测 v 的所有邻接顶点
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
// 深度优先搜索--递归算法
//   参数： v，指定顶点的下标；*visited，记录顶点访问记录，每个顶点是否被访问过，
以及第几个被访问的；
//          count：用于记录在遍历中已经访问了多少个顶点。mode：区别于上一个签名，
暂时不用
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::DFS(const int v, int *visited, int *count, int mode)
{
    *count = *count + 1;
    vertexType temp = GetValue (v);
    visited[v] = *count;          //顶点 v 作访问标记，且记录是第几个被访问的
    //Visit(temp);  //访问顶点 v, 例如 cout, modify----31 题
    if (mode == 18)
    {
        Visit(temp, 3, visited);//---18 题，本质上就是要的满足逆拓扑排序的次序
    }
    else
```

```
        Visit(temp, 2, visited);//---24 题，本质上就是要的满足拓扑排序的次序

    int w = GetFirstNeighbor (v);   //w 是返回的顶点的下标
    while ( w != -1 )          //若顶点 w 存在
    {
        InDegree[w]--;
        if ( !InDegree[w] )
            DFS (w, visited, count, mode);
        w = GetNextNeighbor (v, w);
    }  //重复检测 v 的所有邻接顶点
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
// 深度优先搜索--递归算法
//   参数： v，指定顶点的下标；*visited，记录顶点的最早开始时间或最晚开始时间；

//          mode：区别于上一个签名，且 mode 为 2 时，计算事件的最早开始时间；mode 为 3 时，计算事件的最晚开始时间。
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType, arcType>::DFS(const int v, arcType *visited, int mode)
{
    ArcNode<arcType> *p = NULL;

    switch (mode)
    {
        case 2:
            p = VertexesTable.data[v].firstarc;
            while (p!= NULL)
            {
                if (visited[v] + p->weight > visited[p->adjvex])
                    visited[p->adjvex] = visited[v] + p->weight;
                DFS (p->adjvex, visited, mode);
                p = p->nextarc;
            }
        break;

        case 3:
            p = VertexesTable.data[v].firstarc;
            while (p != NULL)
            {
                DFS(p->adjvex, visited, mode);      //若有后继结点，继续访问
```

```
                if (visited[v] > visited[p->adjvex] - p->weight) //由其后继结点的最晚开始时间，考虑修改其最晚开始时间
                    visited[v] = visited[p->adjvex] - p->weight;
                p = p->nextarc;
            }
        break;
    }
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//深度优先遍历
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType, arcType>::DFTraverse()
{
    int i, n = NumberOfVertexes() ;   //取图的顶点个数
    int * visited = new int [n]; //定义访问标记数组 visited
    int count = 0, block = 0;
    for ( i = 0; i < n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    //对图中的每一个顶点进行判断
    for ( i = 0; i < n; i++ ) {
        if (!visited [i])
        {
            DFS (i, visited, &count);
            block ++;
            cout << "||||以上为第" << block << "个连通分量！" << endl;
        }
    }
    cout << "深度优先遍历结果如上，该图有" << block << "个连通分量！" << endl;
    delete [ ] visited;                    //释放 visited
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//广度优先搜索，返回搜索到的顶点个数
//   参数： v,指定顶点的下标;*visited,记录顶点访问记录,即每个顶点是否被访问过;
//          count：用于记录在遍历中已经访问了多少个顶点。
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
```

```
arcType>::BFS(int v, int *visited, int *count)
{
    int w;
    LinkQueue <int> q;  //定义队列 q
    Visit(GetValue (v));  //访问顶点 v
    visited[v] = 1;          //顶点 v 作已访问标记
    *count += 1;
    q.EnQueue (v);  //顶点 v 进队列 q
    while ( !q.IsEmpty ( ) )
    {
        v = q.DeQueue ( );       //否则, 队头元素出队列
        w = GetFirstNeighbor (v);
        while ( w != -1 )      //若邻接顶点 w 存在
        {
            if ( !visited[w] )   //若该邻接顶点未访问过
            {
                Visit(GetValue (w));  //访问顶点 w
                visited[w] = 1;       //顶点 w 作已访问标记
                *count += 1;
                q.EnQueue (w);  // w 进队列 q
            }
            w = GetNextNeighbor (v, w);
       }    //重复检测 v 的所有邻接顶点
    }    //外层循环, 判队列空否

}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//广度优先遍历
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::BFTraverse()
{
    int i, n = NumberOfVertexes() ;   //取图的顶点个数
    int * visited = new int [n]; //定义访问标记数组 visited
    int count = 0, block = 0;
    for ( i = 0; i < n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    //对图中的每一个顶点进行判断
    for ( i = 0; i < n; i++ ) {
        if (!visited [i])
        {
```

```
            BFS (i, visited, &count);
            block ++;
            cout << "||||以上为第" << block << "个连通分量！" << endl;
        }
    }
    cout << "广度优先遍历结果如上，该图有" << block << "个连通分量！" << endl;
    delete [ ] visited;                    //释放 visited
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//  Prim 普里姆算法求最小生成树
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::Prim(vertexType temp)
{
    int numV = VertexesTable.Length();
    int i, j, v;
    MSTArcNode<vertexType, arcType> *closearc = new MSTArcNode<vertexType,
arcType>[numV];
    MinSpanTree<vertexType, arcType> result(numV);
    MSTArcNode<vertexType, arcType> e;
    arcType min;

    if (numV == 0)
    {
        cout << "空图，无须处理！" << endl;
        return;
    }
    if (NumberOfArcs() == 0)
    {
        cout << "零图，无须处理！" << endl;
        return;
    }
    if (weightGraph == 1)
    {
        cout << "非带权图，没有最小代价的概念，无须处理！" << endl;
        return;
    }

    //初始化
    j = GetVertexPos(temp); //得到起始结点的序号
    closearc[j].SetWeight(NULL);
```

```
closearc[j].SetAdiverFirst(NULL);
for (i = 0; i < numV; i++)
{
    if (i != j)
    {
        closearc[i].SetAdiverFirst(GetValue(j));
        closearc[i].SetWeight(edgeMaxValue);
        ArcNode<arcType> *p = VertexesTable.data[i].firstarc;
        while (p!=NULL)
        {
            if (p->adjvex == j)
            {
                closearc[i].SetAdiverFirst(GetValue(j));
                closearc[i].SetWeight(p->weight);
                p = NULL;
            }
            else
                p = p->nextarc;
        }
    } //end of if (i != j)
}

for ( i = 1; i < numV; i++ )  //循环 n-1 次, 加入 n-1 条边
{
     //选取两个邻接顶点分别在 U-V 和 U 且具有最小权值的边---begin
    min = edgeMaxValue;
    v = -1;
    for ( j = 0; j < numV; j++ )
        if (closearc[j].GetWeight() != NULL && closearc[j].GetWeight() < min )
        {
            v = j;
            min = closearc[j].GetWeight();
        }
     //选取两个邻接顶点分别在 U-V 和 U 且具有最小权值的边--end

    if ( v  != -1)    //v == -1 表示再也找不到所求的边
    {
        e.SetAdiverFirst(closearc[v].GetFirst());
        e.SetAdiverSecond(GetValue(v));
        e.SetWeight(closearc[v].GetWeight());
        result.Insert (e);     //把选出的边加入到生成树中
        closearc[v].SetWeight(NULL); //把顶点 v 加入 U 中
        ArcNode<arcType> *p = VertexesTable.data[v].firstarc;
        while (p!=NULL)
```

```
            {
                // 对 U-V 中的每一个顶点考察是否要修改它在辅助数组中的值
                if     ((closearc[p->adjvex].GetWeight()     !=     NULL)     &&
(closearc[p->adjvex].GetWeight() > p->weight))
                {
                    closearc[p->adjvex].SetWeight(p->weight);
                    closearc[p->adjvex].SetAdiverFirst(GetValue(v));
                }
                p = p->nextarc;
            }//end of while
        } // end of if
     }  // end of for

    //输出结果
    cout << "按照普里姆算法从顶点" << temp << "开始，生成结果---------->";
    result.Display();/**/
}

////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//迪杰斯特拉算法，参数为顶点下标，即存储位置，不是逻辑位置（逻辑位置减 1）
////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::Dijkstra(int v)
{
    arcType min;
    int n = NumberOfVertexes();     //顶点数
    ArcNode<arcType> *x;
    if (n < 1)
    {
        cout << "没有顶点，为空图，无须求最短路径！";
        return;
    }
    if (v < 0  || v >= n)
    {
        cout << "您选择的单源点起点不存在，无须求最短路径！";
        return;
    }

    int u, i, j, w, tempweight;
    distarc = new arcType[n];
    s = new int[n];
    path = new int[n];
```

```
    for (i = 0; i < n; i++)
    {
        x = GetAdj(v, i);
        if (x)
            distarc[i] = x->weight; //dist 数组初始化
        else
            distarc[i] = edgeMaxValue;
        s[i] = 0;  //i 是否已经在 s 中
        if ( i != v && distarc[i] < edgeMaxValue)
            path[i] = v;
        else
            path[i] = -1;        //path 数组初始化
    }
    s[v] = 1; //顶点 v 加入顶点集合 s
    cout << "您选择的源点为：" << GetValue(v) << endl;
    for ( i = 0; i < n-1; i++ )  //按递增序列球最短路径，共 n-1 步
    {
        cout << "第" << i+1 << "步：";
        min = edgeMaxValue;  //找当前最小值--开始
        u = v;
        for (j = 0; j < n; j++ )
            if ( !s[j] && distarc[j] < min )
            {
                u = j;
                min = distarc[j];
            }       //找当前最小值--结束
        cout << "确定的最小值为顶点" << GetValue(v) << "到顶点" << GetValue(u) <<
"的最短距离，值为：" << min << endl;
        cout << "修改后的当前距离值为：" << endl;
        s[u] = 1;                //将顶点 u 加入集合 S
        for (w = 0; w < n; w++ )    //修改 dist 和 path
        {
            x = GetAdj(u,w);
            if (x)
                tempweight = x->weight;
            else
                tempweight = edgeMaxValue;
               if ( !s[w] && distarc[u] + tempweight < distarc[w] )
               {
                    distarc[w] = distarc[u] + tempweight;
                    path[w] = u;
                }
               if (!s[w])
```

```
                {
                    cout << "源点到顶点（" << GetValue(w) << "）的当前最短距离为：
" << distarc[w];
                    cout <<"，经顶点"<<GetValue(path[w])<<"到达该顶点！"<<endl;
                }
        }/**/
        cout << endl;
  }
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
//贝尔曼—福特算法，参数为顶点下标，即存储位置，不是逻辑位置（逻辑位置减 1）
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::BellmanFord(int v)
{
    int n = NumberOfVertexes();     //顶点数
    //vertexType temp = GetValue(v+1);
    if (n < 1)
    {
        cout << "没有顶点，为空图，无须求最短路径！";
        return;
    }
    if (v < 0  || v > n)
    {
        cout << "您选择的单源点起点不存在，无须求最短路径！";
        return;
    }

    int u, i, k;
    arcType *dist = new arcType[n];
    arcType *distTemp = new arcType[n]; //不能在原数组上进行计算
    //相当于递归排序中的两个表的数据来回交换。因为 dist[k]是在 dist[k-1]的基础上求
得的，因此在更新前必须保持 dist[k-1]的完整性
    int *path = new int[n];

    for (i = 0; i < n; i++)
    {
        if ( i != v )//dist 数组初始化
            dist[i] = edgeMaxValue;
        else
            dist[i] = 0;
```

```
    distTemp[i] = edgeMaxValue; //distTemp 数组初始化
    path[i] = -1;        //path 数组初始化
}

ArcNode<arcType> *p = VertexesTable.data[v].firstarc;
/*i = 0;//原题目错误，修改如下 while（p）
while (p)
{
    while (p->adjvex != i)
        i++;
      dist[i] = p->weight;
      distTemp[i] = p->weight;
      if ( i!=v && dist[i] < edgeMaxValue)
          path[i] = v;
      p = p->nextarc;
}*/
while (p)
{
    dist[p->adjvex] = distTemp[p->adjvex] = p->weight;
    if (dist[p->adjvex] < edgeMaxValue)
        path[p->adjvex] = v;
    p = p->nextarc;
}

for (k = 2; k < n; k++)        //循环次数 n-2
{
    cout << "第" << k-1 << "次计算中被修改过的数据如下：" << endl;
    for (u = 0; u < n; u++)    //每个顶点，n 个
    {
       if (u != v)
       {
            for ( i = 0 ; i < n ; i++)     //根据 distk-1[u]计算 distk[u]
                if ( i != v)
                {
                    p = VertexesTable.data[i].firstarc;
                    while ((p!=NULL) && (p->adjvex != u))
                        p = p->nextarc;
                        if ((p!=NULL) && (p->adjvex == u))
                            if (distTemp[u] > dist[i] + p->weight)
                            {
                                distTemp[u] = dist[i] + p->weight;
                                path[u] = i;
                            }
                    //cout << "源点" << temp << "到顶点" << GetValue(u+1) << "
```

```
的当前最短距离由（" << dist[u] << "）修改为（";
                        //cout << distTemp[u] << "，新的嘴短距离经过顶点" <<
GetValue(path[u]+1) << "到达!" << endl;
                    }   //end of if ( i != v)
           } //end of if (u!=v)
        }   // end of for u
        for (i = 0; i < n; i++)
        {
            dist[i] = distTemp[i];
            cout << dist[i] << ",              ";
        }
        cout << endl;
    } //end of for k
    delete distTemp;
    delete dist;
    delete path;
}

//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//利用迪杰斯特拉算法或贝尔曼—福特算法求单源点最短距离
// 参数 type：为 1 是表示给出的是顶点的序号；为 2 时表示给出的是顶点的名称
//////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::ShortestPath(int type, int begin, vertexType &v)
{
    int indexV = -1;;
    bool r = HavePostiveEdge();
    if (type == 2)
    {
        indexV = GetVertexPos(v);
        if (r)
            Dijkstra(indexV);
        else
            BellmanFord(indexV); //该算法还没有最后验证其运行结果---2010-july-1
    }
    else
    {
        if (r)
            Dijkstra(begin);
        else
            BellmanFord(begin);
    }
```

```
    if (r)
    {
        delete distarc;
        delete path;
    }
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//判断图是否连通
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> bool AdjacencyListGraph<vertexType,
arcType>::IsConnected()
{
    bool r1, r2;
    int i, n = NumberOfVertexes() ;   //取图的顶点个数
    int * visited = new int [n+1]; //定义访问标记数组 visited
    int count;

    for ( i = 0; i <= n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    count = 0;
    DFS (1, visited, &count);
    if (count == n)
        r1 = true;
    else
        r1 = false;

    for ( i = 0; i <= n; i++ )
        visited [i] = 0;  //访问标记数组 visited 初始化
    count = 0;
    BFS (1, visited, &count);
    if (count == n)
        r2 = true;
    else
        r2 = false;

    cout << "按深度优先从第一个顶点开始搜索，判断该图连通性结果：" << r1 << endl;
    cout << "按广度优先从第一个顶点开始搜索，判断该图连通性结果：" << r2 << endl;
    if (r1 != r2)
    {
        cout <<"非常遗憾，采用不同的搜索方法，居然得出连通性判断的不同结果，出问
题了，请检查代码！" << endl;
```

```
        exit(1);
    }
    return r1;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//为拓扑排序算法增加入度数组，构造时未初始化入度数组，该函数进行专门处理
//为关键路径算法增加源点个数 begin 和汇点个数 end 的统计。若只有一个源点和汇点，则 begin 和 end 中分别存放源点和汇点的序号
// 若没有源点和汇点，begin 和 end 的值为-1，若多于一个源点或汇点，begin 和 end 的值为-2。
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType, arcType>::InitialInDegree(int *begin, int *end)
{
    if (graphType==1)
    {
        cout <<"提示：该图是无向图，没有出度入度的概念，无须处理入度数组，故而不处理返回！" << endl;
        return;
    }

    int n = NumberOfVertexes();
    int i;
    int *OutDegree = new int[n];
    InDegree = new int[n];
    ArcNode<arcType> *p;

    for (i = 0; i < n; i++)
    {
        InDegree[i] = 0;
        OutDegree[i] = 0;
    }

    //同时记录入读数组和出度数组（局部变量）
    for (i = 0; i < n; i++)
    {
         p = VertexesTable.data[i].firstarc;
         while (p != NULL)
         {
             OutDegree[i]++;
             InDegree[p->adjvex]++;
```

```
            p = p->nextarc;
        }
    }

    //统计源点个数和汇点个数
    *begin = -1;
    *end = -1;
    for (i = 0; i < n; i++)
    {
        if (InDegree[i] == 0)
            if (*begin == -1)
                *begin = i;
            else
                *begin = -2;
        if (OutDegree[i] == 0)
            if (*end == -1)
                *end = i;
            else
                *end = -2;
    }
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//拓扑排序算法
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::TopologicalSort(int *nCycle)
{
    if (graphType == 1)
    {
        cout <<"提示：该图是无向图，没有出度入度的概念，无须处理入度数组，故而不
处理返回！" << endl;
        return;
    }

    int top = -1;                   //入度为零的顶点栈初始化
    int n = 0, j = 0, t;            //n 为输出的顶点数，初始值为 0
    InitialInDegree(&n, &j);        //初始化入度数组
    n = 0;
    int num = NumberOfVertexes();   //取得顶点的个数
    vertexType temp, temp1;
```

```
    for ( j = 0; j < num; j++ )    //入度为零的顶点进栈--采用静态链栈，目的是利用数组 InDegree 同时作为栈，不需要为栈另辟空间
        if ( InDegree[j] == 0 )
        {
            InDegree[j] = top;
            top = j;
        }

    // 拓扑排序部分----A
    cout << "拓扑排序开始，逐个输出排序序列：";
    while (top !=  -1)        //继续拓扑排序
    {
        j = top;
        top = InDegree [top];
        n++; //  输出的顶点数加一
        //temp = GetValue(j+1);
        temp = GetValue(j);
        cout << temp << ", ";
        t = GetFirstNeighbor(j);    //确定弧尾
        while (t != -1)//扫描以顶点 j 为弧尾的所有弧
        {
            if (--InDegree[t] == 0)//顶点 k 的入度减一，若为 0，进栈
            {
                InDegree[t] = top;
                top = t;
            }
            temp1 = GetValue(t);
            t = GetNextNeighbor(j, t);
        }// end of while
    } // end of while top != -1

    delete InDegree; //入度数组在该过程中被申请，为避免后面的其他算法出错，退出该算法时同时释放申请的这些空间

    if ( n < num )
    {
         cout << "AOV 网络中有回路(有向环)！" << endl;
         *nCycle = 1;
    }
    else
    {
        cout << endl;
        *nCycle = 0;
    }
```

```
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//  采用深度优先搜索的拓扑排序算法
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::DFSTopologicalSort(int *nCycle)
{
    if (graphType == 1)
    {
        cout <<"提示：该图是无向图，没有出度入度的概念，无须处理入度数组，故而不
处理返回！" << endl;
        return;
    }

    int n = 0, j = 0;           //n 为输出的顶点数，初始值为 0
    InitialInDegree(&n, &j);        //初始化入度数组
    n = 0;
    int num = NumberOfVertexes();   //取得顶点的个数
    int * visited = new int [num]; //定义访问标记数组 visited

    for ( j = 0; j < num; j++ )    //访问记录初始化
        visited[j] = 0;

    // 拓扑排序部分----A
    for ( j = 0; j < num; j++ )
        if (!visited[j] && !InDegree[j])
            DFS(j, &visited[0], &n, *nCycle);

    delete InDegree; //入度数组在该过程中被申请，为避免后面的其他算法出错，退出该
算法时同时释放申请的这些空间

    if ( n < num )
    {
         cout << "AOV 网络中有回路(有向环)！" << endl;
         *nCycle = 1;
    }
    else
    {
        cout << endl;
        *nCycle = 0;
    }
```

```
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//    判断是否为有向无环图，是，返回 true；否则，返回 false。
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> bool AdjacencyListGraph<vertexType,
arcType>::IsDAG()
{
    if (graphType != 2)
    {
        cout << "提示：该图是无向图！" << endl;
        return false;
    }
    int nVNum;
    TopologicalSort(&nVNum);
    if (nVNum == 1)
    {
        cout << "提示：该图有环！" << endl;
        return false;
    }
    else
        if (nVNum == 0)
        {
            cout << "提示：该图是有向无环图！" << endl;
            return true;
        }
        else
        {
            cout <<"提示：该图测下来既说有环又说无环，建议调试程序，肯定哪里出问
题啦！" << endl;
            return false;
        }
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//最小生成树的克鲁斯卡尔算法
//和教材上的区别：无须传递参数，在该算法中定义局部变量 result 用于存储最小生成树
//（除非生成树的结果需要继续使用，否则，定义为局部变量演示算法即可）
//测试数据：顶点--abcdef#；边--ab6，ac1，ad4，bc5，be3，cd5，ce6，cf4，df2，ef6；
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
```

```
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType, arcType>::Kruskal()
{
    int numV = NumberOfVertexes();
    int i, u, v;
    MSTArcNodeForHeap<arcType>    **a    =    new    MSTArcNodeForHeap<arcType>*[CurrentNumArcs];
    MSTArcNode<vertexType,    arcType>    *aa    =    new    MSTArcNode<vertexType, arcType>[CurrentNumArcs];
    MinSpanTree<vertexType, arcType> result(numV);
    int count  = 0;
    MSTArcNode<vertexType, arcType> e;
    MSTArcNodeForHeap<arcType> *e1;
    vertexType *b = new vertexType[numV];

    if (numV == 0)
    {
        cout << "空图，无须处理！" << endl;
        return;
    }
    if (NumberOfArcs() == 0)
    {
        cout << "零图，无须处理！" << endl;
        return;
    }
    if (weightGraph == 1)
    {
        cout << "非带权图，没有最小代价的概念，无须处理！" << endl;
        return;
    }

    for (i = 0; i < CurrentNumArcs; i++)
        a[i] = new MSTArcNodeForHeap<arcType>;

    MinSpanTreeHeap<MSTArcNodeForHeap<arcType> *> h(CurrentNumArcs, 1);//第三版本---1
    //将图中所有的边的完整信息置于数组中，依此构造最小堆，用于选择权值最小的边
    ArcNode<arcType> *p;
    for (i = 0; i < numV; i++)
    {
        p = VertexesTable.data[i].firstarc;
        while (p != NULL)
        {
            if ((graphType == 1 && i < p->adjvex) || (graphType == 2))
```

```
            {
                a[count]->SetWeight(GetWeight(GetValue(i),
GetValue(p->adjvex)));
                a[count]->SetID(count);
                h.Insert(a[count]);                    //第三版本---2

                aa[count].SetAdiverFirst(GetValue(i));
                aa[count].SetAdiverSecond(GetValue(p->adjvex));
                aa[count].SetWeight(GetWeight(GetValue(i),
GetValue(p->adjvex)));

                count++;
            }
            p = p->nextarc;
        }

        b[i] = GetValue(i);
    }
    /*
    //第二版本，能够编译运行，但构造的初始堆是按照输入顺序（即指针地址）
    //调用的是父类的 FiltDown 函数，而不是子类的，但插入、删除却正确
    //关于派生类与父类间的构造函数的细节语法，需要再详细查看。改第三版本
    //MinSpanTreeHeap<MSTArcNodeForHeap<arcType> *> h(a, count, 1);
    */
    cout << "所建立的最小堆为：";
    h.out();
    /*
    //第一版本：两个类型抽象，与最小堆的定义不太吻合，为了避免重新定义堆，修改为
第二版本
//  MinSpanTreeHeap<MSTArcNode<vertexType, arcType> > h(a, count, 1);
*/

    //根据所有顶点构造并查集
    UFSets<vertexType> uf(b, NumberOfVertexes());

    //生成最小生成树，共 numV-1 条边，故循环 numV 次
    i = 1;

    while (i < numV)
    {
        e1 = h.DeleteTop();
        cout << "Check:" << e1->GetID() << "," << e1->GetWeight() << "; ";
        e = aa[e1->GetID()];
        u = uf.Find(e.GetFirst());
```

```
        v = uf.Find(e.GetSecond());
        if (u != v) //u,v 不在同一个并查集中，即加入(u, v)不构成环
        {
            uf.Union(u, v);
            result.Insert(e);
            i++;
        }
    }

    //输出结果
    cout << "按照克鲁斯卡尔算法生成结果---------->";
    result.Display();
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
//弗洛伊德(Floyd)算法
////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::Floyd()
{
    int n = NumberOfVertexes();     //顶点数
    arcType a[MaxVertexes][MaxVertexes];
    int i, j, k;
    int path[30][30];
    ArcNode<arcType> *p;

    //矩阵 a(-1)与 path(-1)初始化 0
    for (i = 0; i < n; i++ )
        for (j = 0; j < n; j++ )
        {
            if (i == j)
                a[i][j] = 0;
            else
                a[i][j] = edgeMaxValue;
            path[i][j] = -1;    //从 i 到 j 没直接的路径（弧）
        }   // end of for
    //矩阵 a(-1)与 path(-1)初始化 1
    for (i = 0; i < n; i++ )
    {
        p = VertexesTable.data[i].firstarc;
        while (p != NULL)
        {
```

```
            a[i][p->adjvex] = p->weight;
            path[i][j] = i;   //从 i 到 j 有直接的路径（弧）
            p = p->nextarc;
        }
    }

    //产生 A(k)及 path(k)
    for (k = 0; k < n; k++ )
    {
        cout << "第" << k+1 << "次计算：" << endl;
        for ( i = 0; i < n; i++ )
        {
            for ( j = 0; j < n; j++ )
            {
             /*  if ( a[i][k] + a[k][j] < a[i][j] ) //缩短路径长度, 经过 k 到 j,
权值为正数，没有问题。权值为负数，会出现
                {                                   //表示无穷的最大值+一个负数
< 原来表示无穷的最大值*/
                if (a[i][k] != edgeMaxValue && a[k][j] != edgeMaxValue && a[i][k]
+ a[k][j] < a[i][j] )
                {
                    a[i][j] = a[i][k] + a[k][j];
                    path[i][j] = path[k][j];
                } // end of if
                cout << a[i][j] << ",            ";
            }
            cout << endl;
        }
    }
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
/*
```

7.14　试基于图的深度优先搜索策略写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 vi 到顶点 vj 的路径(i!=j)。
注意：算法中涉及的图的基本操作必须在此存储结构上实现。

7.15　试基于图的广度优先搜索策略编写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 vi 到顶点 vj 的路径(i!=j)。
注意：算法中涉及的图的基本操作必须在此存储结构上实现。

参数：op 为 D 或 d，表示采用深度优先搜索算法；为 B 或 b 表示采用广度优先搜索算法

```
*/
template<class vertexType, class arcType> int AdjacencyListGraph<vertexType,
arcType>::ExistPath(vertexType vi, vertexType vj, int op, int l)
{
    int i, j, n, k, count;
    //取顶点 vi 和 vj 在图中的序号
    i = GetVertexPos( vi);
    j = GetVertexPos(vj);
    if ( i == -1|| j == -1 )
    {
        cout << "您提供的结点不存在！无法继续判断！" << endl;
        return -1; //顶点 vi 或 vj 不存在，返回-1。
    }
    if (i == j)
    {
        cout<<"顶点输入错误，不能为同一顶点！数据结构算法不考虑顶点自身的关系！"
<< endl;
        return -1;
    }
    n = NumberOfVertexes() ; //取图的顶点个数
    int * visited = new int [n]; //定义访问标记数组 visited
    for (k = 0; k < n; k++ )
        visited[k] = 0; //访问标记数组 visited 初始化

    if (l != -100)
        return DFS_ExistPath(vi, vj, l, visited);
    if (op == 'D' || op == 'd')
    {
        if (op == 'D')
            DFS (i, visited, &count) ; //从 vi 开始进行深度优先搜索
        else
            return DFS_ExistPath(vi, vj, visited);
    }
    else
        if  (op == 'B' || op == 'b')
        {
            if  (op == 'B')
                BFS (i, visited, &count) ; //从 vi 开始进行广度优先搜索
            else
                return BFS_ExistPath(vi, vj);
        }
        else
        {
            cout <<"您选择的方法既不是深度优先也不是广度优先搜索方法，本题不提供
```

```
其他搜索方式！" << endl;
            return -1;
        }
    i = visited[j] ; //保存结果以便于返回
    delete [ ] visited; //释放 visited
    return i;//返回结果
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
/*

7.14    试基于图的深度优先搜索策略写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 vi 到顶点 vj 的路径(i!=j)。
注意：算法中涉及的图的基本操作必须在此存储结构上实现。[解法二]

*/
template<class vertexType, class arcType> int AdjacencyListGraph<vertexType, arcType>::DFS_ExistPath(vertexType vi, vertexType vj, int visited[])
{
    int i, j, n, k;
    ArcNode<arcType> *p;
    //取顶点 vi 和 vj 在图中的序号
    i = GetVertexPos( vi);
    j = GetVertexPos(vj);
    if ( i==-1|| j==-1 )
    {
        cout << "您提供的结点不在！无法继续判断！" << endl;
        return -1; //顶点 vi 或 vj 不存在，返回-1。
    }
    if (i==j)
    {
        cout<<"顶点输入错误，不能为同一顶点！数据结构算法不考虑顶点自身的关系！" << endl;
        return -1;
    }
    n = NumberOfVertexes() ; //取图的顶点个数

    visited[i] = 1;
    for( p = VertexesTable.data[i].firstarc; p; p = p->nextarc) {
        k = p->adjvex;
        if(!visited[k] && DFS_ExistPath(GetValue(k),vj, visited)) //i 下游的顶点到 j 有路径
            return 1;
```

```
    } // end of for

    return 0;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
/*

7.15    试基于图的广度优先搜索策略编写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 vi 到顶点 vj 的路径(i!=j)。
注意：算法中涉及的图的基本操作必须在此存储结构上实现。[解法二]

*/
template<class vertexType, class arcType> int AdjacencyListGraph<vertexType, arcType>::BFS_ExistPath(vertexType vi, vertexType vj)
{
    int i, j, n, k, u;
    ArcNode<arcType> *p;
    LinkQueue<int> Q;

    //取顶点 vi 和 vj 在图中的序号
    i = GetVertexPos( vi);
    j = GetVertexPos(vj);
    if ( i==-1|| j==-1 )
    {
        cout << "您提供的结点不在！无法继续判断！" << endl;
        return -1; //顶点 vi 或 vj 不存在，返回-1。
    }
    if (i==j)
    {
        cout<<"顶点输入错误，不能为同一顶点！数据结构算法不考虑顶点自身的关系！" << endl;
        return -1;
    }
    n = NumberOfVertexes() ; //取图的顶点个数
    int * visited = new int [n]; //定义访问标记数组 visited
    for (k = 0; k < n; k++ )
        visited[k] = 0; //访问标记数组 visited 初始化

    visited[i] = 1;
    Q.EnQueue(i);
    while( !Q.IsEmpty())
    {
```

```
        u = Q.DeQueue();
        visited[u] = 1;
        for (p = VertexesTable.data[u].firstarc; p; p = p->nextarc) {
            k = p->adjvex;
            if (k == j)
                return 1;
            if (!visited[k])
                Q.EnQueue(k);
        } //end of for
    } // end of while

    delete [ ] visited; //释放 visited
    return 0;
}

////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////
/*
```

7.19　试写一算法，判别以邻接表方式存储的有向图中是否存在由顶点 vi 到顶点 vj 的长度为 L 的简单路径(i!=j)。
注意：算法中涉及的图的基本操作必须在此存储结构上实现。[基于图的深度优先搜索策略]

```
*/
template<class vertexType, class arcType> int AdjacencyListGraph<vertexType,
arcType>::DFS_ExistPath(vertexType vi, vertexType vj, int L, int visited[])
{
    int i, j, k, newl;
    ArcNode<arcType> *p;
    //取顶点 vi 和 vj 在图中的序号
    i = GetVertexPos( vi);
    j = GetVertexPos(vj);
    if ( i == -1|| j == -1 )
    {
        cout << "您提供的结点不在！无法继续判断！" << endl;
        return -1; //顶点 vi 或 vj 不存在，返回-1。
    }
    if ( i == j && L == 0)
        return 1; //找到了一条路径,且长度符合要求
    if (L < 0)
    {
        cout << "路径长度小于 0，无法继续查找！" << endl;
        return -1; //长度小于 0，返回-1。
    }
```

```
    visited[i] = 1;
    for( p = VertexesTable.data[i].firstarc; p; p = p->nextarc)
    {
        k = p->adjvex;
        if(!visited[k])
        {
            if (weightGraph == 2)
                newl = L - GetWeight(i, k);
            else
                newl = L - 1;
            if ( newl < 0)
                return -1;
            if (DFS_ExistPath(GetValue(k),vj, newl, visited) == 1)
                return 1;
        }
    } // end of for

    visited[i] = 0; //本题允许曾经被访问过的结点出现在另一条路径中

    return 0;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
/*
```

7.25　若有向无环图中存在一个顶点 r，如果在 r 和图中其他所有顶点之间均存在由 r 出发的有向路径，则称该 DAG 有根。

试编写求有向无环图中根的算法。有根，返回根的序号，否则，返回-1。图类型不正确，返回-2

注意：　该算法要求不能有环，否则会发生误判。

```
*/
template<class vertexType, class arcType> int AdjacencyListGraph<vertexType, arcType>::GetRoot()
{
    if (IsDAG() == false)
    {
        cout << "不是有向无环图，根的概念无法正确给出。退出！" << endl;
        return -2;
    }

    int i, j, n, flag, visited[20];
```

```
    n = VertexesTable.Length();          //取得图的顶点数
    for (i = 0; i < n; i++) // 对每一个顶点，判断其是否为根
    {
        for (j = 0; j < n; j++)
            visited[j] = 0; //访问数组清零
        flag = 0;
        DFS(i, &visited[0], &flag);
        if (flag == n) //能访问到所有顶点, i 是根
            return i;
        //思考：如果需要求得所有的根，如何修改程序？
    }
    return -1;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////////////////
/*

7.28   利用深度优先遍历有向图实现求关键路径算法。

*/
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::DFSCriticalPath()
{
    if (IsDAG() == false)
    {
        cout << "不是有向无环图，无法求关键路径的。会死循环的。退出！" << endl;
        return;
    }

    int n = 0, k = 0;
    InitialInDegree(&n, &k);
    //源点汇点不存在或多于一个，不处理------可以处理，但不提倡
    if (n < 0)
    {
        cout << "该图的源点";
        if (n == -1)
            cout << "不存在，没有源点的图无法求关键路径，";
        else
            if (n == -2)
                cout << "不是一个，多个源点的图不考虑，";
            else
                cout << "求源点时出错了，请检查程序代码，";
```

```
        cout << "因此，没有处理，直接返回了！" << endl;
        return;
    }
    if (k < 0)
    {
        cout << "该图的汇点";
        if (k == -1)
            cout << "不存在，没有汇点的图无法求关键路径，";
        else
            if (k == -2)
                cout << "不是一个，多个汇点的图不考虑，";
            else
                cout << "求汇点时出错了，请检查程序代码，";
        cout << "因此，没有处理，直接返回了！" << endl;
        return;
    }

    n = VertexesTable.Length();          //取得图的顶点数
    arcType * Ve = new int [n]; //定义事件最早开始时间数组 Ve
    arcType * Vl = new int [n]; //定义事件最晚开始时间数组 Vl
    arcType info = 0;
    for (k = 0; k < n; k++ )
    {
        Ve[k] = 0; //事件最早开始时间数组 Ve 初始化
        Vl[k] = 0; //事件最晚开始时间数组 Ve 初始化
    }

    DFS(0, &Ve[0], 2);       //深度优先搜索，计算出各个事件【顶点】的最早开始时间

    //各个事件【顶点】的最晚开始时间的初始化----第一版参考答案把 Vl 的初始化放在
DFS 递归内是不正确的，因为递归可能多次“经过”汇点
    for (k = 0; k < n; k++ )
        if (VertexesTable.data[k].firstarc == NULL)     //是汇点
            info = Ve[k];
    for (k = 0; k < n; k++ )
        Vl[k] = info;

    DFS(0, &Vl[0], 3);  //深度优先搜索，计算出各个事件【顶点】的最晚开始时间

    //输出所有的关键活动
    for (k = 0; k < n; k++ )
        cout << "顶点" << GetValue(k) << "的最早开始时间为" << Ve[k] << "，最晚开
始时间为" << Vl[k] << endl;
```

```
    for (k = 0; k < n; k++ )
        if (Ve[k] == Vl[k])
            cout << GetValue(k) << ", ";
    cout << endl;

    delete  [ ] Ve;
    delete  [ ] Vl;
}
```