```
////////////////////////////////////////////////////////////////////////////////
//////////////////////////
//   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 ( 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, 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;
}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//深度优先搜索
//   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)
{

}


//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//深度优先遍历
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::DFTraverse()
{

}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//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)
{

}

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

```
//广度优先遍历
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::BFTraverse()
{

}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//  Prim 普里姆算法求最小生成树
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
template<class vertexType, class arcType> void AdjacencyListGraph<vertexType,
arcType>::Prim(vertexType temp)
{

}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//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)
{

}

//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
//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)
{

}

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

```
//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
//////////////////////////////////////////////////////////////////////////////
//////////////////////////////////////////////
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()
{

}

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

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。

*/
```
template<class vertexType, class arcType> int AdjacencyListGraph<vertexType, arcType>::ExistPath(vertexType vi, vertexType vj, int op, int l)
{

}

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

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[])
{

}

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

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)
{
```

```
}

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

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;

}
```