```
/* 邻接表存储 - Kruskal最小生成树算法 */

/*-------------------- 顶点并查集定义 --------------------*/
typedef Vertex ElementType; /* 默认元素可以用非负整数表示 */
typedef Vertex SetName;     /* 默认用根结点的下标作为集合名称 */
typedef ElementType SetType[MaxVertexNum]; /* 假设集合元素下标从0开始 */

void InitializeVSet( SetType S, int N )
{ /* 初始化并查集 */
    ElementType X;

    for ( X=0; X<N; X++ ) S[X] = -1;
}

void Union( SetType S, SetName Root1, SetName Root2 )
{ /* 这里默认Root1和Root2是不同集合的根结点 */
    /* 保证小集合并入大集合 */
    if ( S[Root2] < S[Root1] ) { /* 如果集合2比较大 */
        S[Root2] += S[Root1];     /* 集合1并入集合2  */
        S[Root1] = Root2;
    }
    else {                        /* 如果集合1比较大 */
        S[Root1] += S[Root2];     /* 集合2并入集合1  */
        S[Root2] = Root1;
    }
}

SetName Find( SetType S, ElementType X )
{ /* 默认集合元素全部初始化为-1 */
    if ( S[X] < 0 ) /* 找到集合的根 */
        return X;
    else
        return S[X] = Find( S, S[X] ); /* 路径压缩 */
}

bool CheckCycle( SetType VSet, Vertex V1, Vertex V2 )
{ /* 检查连接V1和V2的边是否在现有的最小生成树子集中构成回路 */
    Vertex Root1, Root2;

    Root1 = Find( VSet, V1 ); /* 得到V1所属的连通集名称 */
    Root2 = Find( VSet, V2 ); /* 得到V2所属的连通集名称 */

    if( Root1==Root2 ) /* 若V1和V2已经连通，则该边不能要 */
        return false;
    else { /* 否则该边可以被收集，同时将V1和V2并入同一连通集 */
        Union( VSet, Root1, Root2 );
        return true;
    }
}
/*-------------------- 并查集定义结束 --------------------*/

/*-------------------- 边的最小堆定义 --------------------*/
void PercDown( Edge ESet, int p, int N )
{ /* 改编代码4.24的PercDown( MaxHeap H, int p )    */
  /* 将N个元素的边数组中以ESet[p]为根的子堆调整为关于Weight的最小堆 */
    int Parent, Child;
    struct ENode X;

    X = ESet[p]; /* 取出根结点存放的值 */
    for( Parent=p; (Parent*2+1)<N; Parent=Child ) {
        Child = Parent * 2 + 1;
        if( (Child!=N-1) && (ESet[Child].Weight>ESet[Child+1].Weight) )
            Child++;  /* Child指向左右子结点的较小者 */
        if( X.Weight <= ESet[Child].Weight ) break; /* 找到了合适位置 */
        else  /* 下滤X */
            ESet[Parent] = ESet[Child];
    }
    ESet[Parent] = X;
}

void InitializeESet( LGraph Graph, Edge ESet )
{ /* 将图的边存入数组ESet，并且初始化为最小堆 */
    Vertex V;
    PtrToAdjVNode W;
    int ECount;

    /* 将图的边存入数组ESet */
    ECount = 0;
    for ( V=0; V<Graph->Nv; V++ )
        for ( W=Graph->G[V].FirstEdge; W; W=W->Next )
            if ( V < W->AdjV ) { /* 避免重复录入无向图的边，只收V1<V2的边 */
```

```
                ESet[ECount].V1 = V;
                ESet[ECount].V2 = W->AdjV;
                ESet[ECount++].Weight = W->Weight;
            }
    /* 初始化为最小堆 */
    for ( ECount=Graph->Ne/2; ECount>=0; ECount-- )
        PercDown( ESet, ECount, Graph->Ne );
}

int GetEdge( Edge ESet, int CurrentSize )
{ /* 给定当前堆的大小CurrentSize，将当前最小边位置弹出并调整堆 */

    /* 将最小边与当前堆的最后一个位置的边交换 */
    Swap( &ESet[0], &ESet[CurrentSize-1]);
    /* 将剩下的边继续调整成最小堆 */
    PercDown( ESet, 0, CurrentSize-1 );

    return CurrentSize-1; /* 返回最小边所在位置 */
}
/*-------------------- 最小堆定义结束 --------------------*/


int Kruskal( LGraph Graph, LGraph MST )
{ /* 将最小生成树保存为邻接表存储的图MST，返回最小权重和 */
    WeightType TotalWeight;
    int ECount, NextEdge;
    SetType VSet; /* 顶点数组 */
    Edge ESet;    /* 边数组 */

    InitializeVSet( VSet, Graph->Nv ); /* 初始化顶点并查集 */
    ESet = (Edge)malloc( sizeof(struct ENode)*Graph->Ne );
    InitializeESet( Graph, ESet ); /* 初始化边的最小堆 */
    /* 创建包含所有顶点但没有边的图。注意用邻接表版本 */
    MST = CreateGraph(Graph->Nv);
    TotalWeight = 0; /* 初始化权重和     */
    ECount = 0;      /* 初始化收录的边数 */

    NextEdge = Graph->Ne; /* 原始边集的规模 */
    while ( ECount < Graph->Nv-1 ) {  /* 当收集的边不足以构成树时 */
        NextEdge = GetEdge( ESet, NextEdge ); /* 从边集中得到最小边的位置 */
        if (NextEdge < 0) /* 边集已空 */
            break;
        /* 如果该边的加入不构成回路，即两端结点不属于同一连通集 */
        if ( CheckCycle( VSet, ESet[NextEdge].V1, ESet[NextEdge].V2 )==true ) {
            /* 将该边插入MST */
            InsertEdge( MST, ESet+NextEdge );
            TotalWeight += ESet[NextEdge].Weight; /* 累计权重 */
            ECount++; /* 生成树中边数加1 */
        }
    }
    if ( ECount < Graph->Nv-1 )
        TotalWeight = -1; /* 设置错误标记，表示生成树不存在 */

    return TotalWeight;
}
```